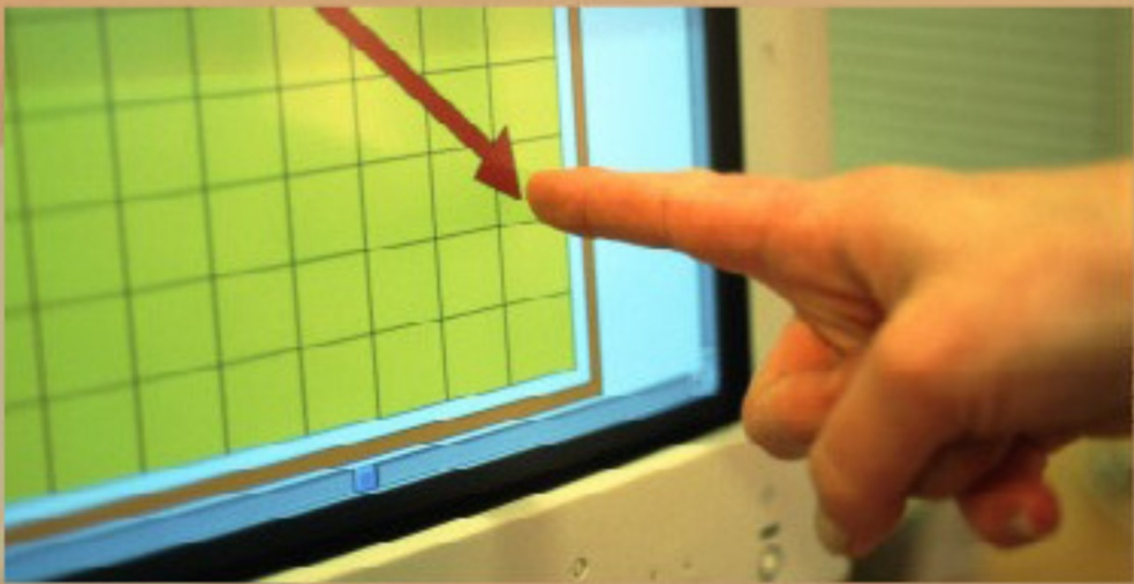
Python for Informatics

Exploring Information

中文版



Charles Severance

范炜 张功卫 王骏译

目錄

介绍	0
译者序	0.1
作者序	0.2
为什么要学习编程	1
变量、表达式与语句	2
条件执行	3
函数	4
迭代	5
字符串	6
文件	7
列表	8
字典	9
元组	10
正则表达式	11
网络编程	12
Web Services	13
数据库与结构化查询语言SQL	14
数据可视化	15
常见任务自动化处理	16
附录A	17
附录B	18
附录C	19

介绍

Python for Informatics（中文版）《信息管理专业Python教程》针对编程初学者，介绍如何编程，使用Python进行数据处理与数据可视化。

本书旨在透过数据探索的镜头，向学生介绍编程与计算思维。Python可视为远比电子表格强大的问题解决工具。Python是一种易于使用与学习的编程语言，在Windows、Mac与Linux计算机上都可以免费获取。

本书采用“署名-非商业性使用-相同方式共享”许可形式，提供各种电子格式的免费下载。另外，本书配套一门免费在线自学课程(http://www.pythonlearn.com)。

书中所有内容材料是开放的，可用于改编。

翻译说明

得益于开源与共享精神，根据知识共享协议CC-BY-SA，经由作者授权，这本书的简体中文版翻译与开放出版得以顺利完成。

翻译工作由范炜牵头，得到四川大学公共管理学院信息管理综合实验室的大力支持，胡康林实验师审校了本书并提出宝贵意见，信息管理与信息系统专业两位本科生参与了部分章节的翻译与校对。

- 张功卫 翻译第1、3、6、8、9、10章及附录。
- 王　骏 翻译第2、4、5、7章。
- 范　炜 翻译第11-16章，负责统稿与定稿。

由于精力与水平有限，书中翻译错误在所难免。欢迎各位读者批评指正，我们会汲取建议，及时进行修订。

勘误贡献

感谢吴志暹纠正了文中的一些笔误和提出Gitbook Markdown语法解析问题。

译者序

翻译普遍被认为出力不讨好，这次我把翻译当成一件专业情结的活来干了。

本序由4个问题和1个阅读说明组成，希望能引起相关读者的共鸣。

为什么要冠以信息管理专业之名？

Informatics在中文语境下是个神秘的术语，有情报学、信息学等说法，但这两个名称说实话都不合适。国内的情报学不是搞间谍的，信息学是搞通讯编码的。在信息的世界里，美丽的误会无处不在。通俗讲，Informatics属于信息管理大范畴，是信息管理活动的高级阶段，通过数据分析与挖掘，提供对管理与决策有价值的信息（即情报）。

信息管理听起来非常有魅惑力。从早期一波波的数字化浪潮到当代的大数据，信息管理专业无可争议地处在信息化时代变革的先锋阵营，一直与各类型数据（这里可视为信息的载体）打交道。这个专业的业务主线是围绕数据（信息）的创建、采集、传递、加工与分析利用等。

为什么信息管理专业要学习编程？

信息管理离不开信息技术，倒不如说各类信息技术是为解决信息管理业务中的实际问题而被发明的。依据此理，学习编程是用技术方法与工具来解决信息管理中的具体问题和提高工作效率。这看似有道理，但现实中编程让这个专业的学生感到困惑和畏惧，常常与计算机专业的编程课混淆侧重，学起来既无趣又痛苦。

为什么是**Python**？

C语言是大多数信息管理专业开设的第一门编程课。毋庸置疑，学习C语言首先从编程思想上武装了初学者。然而，信息管理专业的学生未来并不打算从事编程语言本身的研究，更多想要用编程工具来解决实际问题。另外，信息管理专业课程体系里并没有过多的编程深入课程，编程入门与应用衔接比较薄弱或直接偏向了信息系统开发（管理信息系统方向），学生从C语言向其他眼花缭乱的高级编程语言转变感到困难，导致学习的成就感普遍不高。学了编程，拿着一张全国计算机等级考试C语言证书，没有什么实质意义，运用不起来是最大的问题所在。

Python语言的简洁、功能全面性与易于学习等特点在编程语言入门级广泛流行，近些年可以说是深得民心。除了语言本身的优点之外，Python这门语言非常适合信息管理专业，原因在于它的功能性全面渗透到信息管理的各个业务环节，如书中介绍的文本处理、数据采集、数据库存储与调用、Web Services等主题经由Python做到了很好的知识串联。可以说，Python提供了编程学习与技术应用贯通的统一化平台。

这本书将Python语言的讲解与具体数据管理问题相结合，在一定程度做到了学会即用，是目前比较少见的适合信息管理专业的编程入门教材。

为什么要翻译这本书？

开篇也提到这是带有专业情结的事儿。过去，我曾是国内最早一批信息管理与信息系统专业的学生，现在，我是这个专业的一名教师。十多年来与这个专业共同成长，自身也经历了对编程的种种迷茫与困惑。随着信息技术的快速变化，教学内容与手段的与时俱进也显得非常必要。工作之后，我看到了Python带来打通信息管理专业编程学习症结的可能性，开始翻阅各类Python书籍。眼前的这本书让人眼前一亮，我非常认同本书作者的教学理念，他的努力让信息管理专业学生能够较为轻松愉悦地学习一门功能强大的应用型高级编程语言。

通过翻译这本书，希望更多信息管理专业的学生能够找到编程入门的钥匙，将Python作为通向数据管理与分析技能养成的一座桥梁，少走一些弯路。对我而言，更多是在未来的教学中引入Python内容，设计出适合国内信息管理专业的数据技能类应用型课程。

如何阅读这本书？

虽然阅读这本书不需要什么技术基础，但最好具备操作系统、数据库与互联网的一些基础知识。

本书内容分为两大部分：第1-10章涵盖Python语言的基础知识；第11-16章是Python在数据管理与分析中的应用。书中的所有示例与代码均可免费下载，建议边读书边操作，手脑并用收获才会更多。12-15章的程序代码相对复杂，如果感到自学难度大，可先按照书中讲解，执行源代码，输出结果查看效果，走通之后再回过头来慢慢研究代码细节。

工欲善其事必先利其器。正如本书的副标题，希望Python能成为你专业学习道路上探索信息世界的一把利器。

致谢

首先要感谢美国密西根大学信息学院Charles Severance教授的慷慨支持，使得本书得以顺利自由出版。

技术翻译不是简单的文字工作，费时费心费力，事无巨细也难以做到不出错。从项目启动到完成历时近3个月。感谢这段时间以来为该项目提供支持和关心我的所有人，没有你们的鼓励与帮助，单凭我个人无法完成这项工作，这段经历令人难忘。

感谢一路有你们！

范 炜
四川大学信息管理技术系
2015年6月于雪城

作者序

信息管理专业Python教程：一本混编的开源书

在学术界，科研人员不断被洗脑的是“发表或消亡”，从头开始创造，做出原始创新。本书不是原始创新，而是对《像计算机科学家一样思考Python》（以下简称为《思考Python》）（Allen B. Downey, Jeff Elkner等著）这本书的一次混编实验。

2009年12月，我在密西根大学准备讲授“SI502-网络编程”这门课（连续第五学期开设），决定开始编写一本侧重于数据探索，而不是理解算法与抽象理论的Python教材。我的SI502课程目标是培养学生使用Python的终生数据处理技能。我的学生中间很少有人打算成为计算机程序员。相反，他们的职业规划是图书馆员、经理、律师、生物学家、经济学家等。他们希望在所从事的行业中能熟练运用技术。

我似乎找不到适合这门课的Python教材，侧重以数据为中心的，因此我决定要编写这样一本书。我打算利用假期时间从头编写这本书，所幸，在放假前三周的教职工会议上，Atul Prakash博士带给我一本书《思考Python》，他在那个学期使用这本书讲授Python课程。这是一本很棒的计算机专业教材，特点在于篇幅短、直白的解释以及易于学习。

本书的整体结构已经调整为尽可能快速地解决数据分析问题，包括一系列从基础到高阶的可运行的数据分析示例与练习。

第2-10章与《思考Python》内容类似，主要区别如下：面向数值的示例与练习被面向数据的练习取代；主题的顺序按照循序渐进原则，从简单数据处理到复杂的数据分析解决方案；一些主题，例如try与except调整到前面，作为第3章 “条件”的部分内容。由于函数比较抽象，没有过早出现，当需要处理程序的复杂性时才予以介绍。几乎所有用户定义的函数都从第4章的例代码与练习中移除了。本书不会出现“递归”[1]这个术语。

第1章和第11-16章的内容都是全新的，主要介绍Python的现实应用，包括一些用于数据分析的Python简单示例，例如，搜索与解析的正则表达式，计算机上任务的自动化执行，通过网络检索数据，采集网页数据，Web Services的使用，解析XML与JSON数据，使用结构化查询语言SQL进行数据库的创建与使用等。

本书做出这些调整的最终目标是，从计算机专业视角转向信息管理专业视角，仅仅涵盖技术入门课程必需的主题，即使选课的学生并不打算成为专业的程序员也会感到有用。

对这本书感兴趣的学生想要进一步探索的话，应该阅读Alle B. Downey的《思考Python》。由于两本书的内容存在许多重叠，学生能从那本书中快速掌握更多技术编程技能与提升算法思考能力。另外，两本书的写作风格相似，读完这本书之后应该能很容易通读《思考Python》。

《思考Python》这本书的版权所有者Allen授权我对书中内容的版权做了修改，将本书中保留的他书中的内容许可从GNU自由文档许可变更为最近流行的知识共享——以相同方式共享（CC-BY-SA）协议。这一做法符合开放文档许可从GFDJ到CC-BY-SA的转变趋势（如，维基百科的授权方式）。CC-BY-SA许可保留了图书最显著的版权传统，同时又允许新作者直接重用书中他们觉得合适的内容。

我觉得这本书是教学资料开放的一个典范，对未来的教育而言非常重要。感谢Alleb B. Downey和剑桥大学出版社做出的前瞻决定，让这本书可以在开放版权下出版。希望他们对我努力的结果感到欣慰，我也希望读者对我们的集体努力感到高兴。

我还要感谢Allen B. Downey与Lauren Cowles在解决本书版权问题时提供的帮助、耐心与指导。

Charles Severance
www.dr-chuck.com
Ann Arbor, MI, USA
2013年9月9号

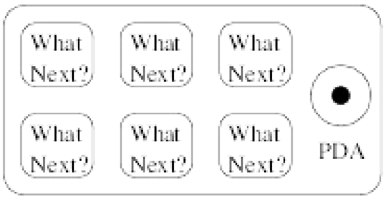
Charles Severance博士是密西根大学信息学院的临床副教授。

> 1. 当然，这里出现的递归除外。 ↩

第1章 为什么要学习编程

编程是一项极具创造性和有益的活动。编程的原因很多，大到为谋生去解决一个困难的数据分析问题，小到因为帮助别人解决一个问题而获得快乐。本书假定每个人都需要知道怎样编程，一旦学会编程，你就会想要用这个新技能做些什么了。

我们的日常生活中计算机无处不在，大到笔记本电脑，小到手机。这些计算机可视为帮助我们打理很多事情的“私人助理”。如今的计算机硬件从根本上是构建在不断回答我们提问的基础上，即“下一步想要做什么？”



程序员在硬件之上添加了操作系统和应用程序，我们手中拿到的成品是一个很有用的个人数字助理（PDA，Personal Digital Assistant），它能够帮我们处理很多不同的事情。

计算机运行速度很快并拥有大量的内存，如果我们知道了如何与计算机沟通，告诉计算机想要它接下来做什么，这对更好地使用计算机很有帮助。如果掌握了计算机沟通语言，就能让计算机根据我们的意愿完成一些重复性工作。有趣的是，计算机能够胜任并且做得很好的工作就是那些经常让我们人类感到无聊、令人头脑麻木的事情。

例如，阅读本章的前三段，找出出现频率最高的词是哪一个，以及这个词总共出现了多少次。尽管你能在短时间内阅读和理解这些文字，但要对它们进行统计就很痛苦了。这类问题不是人的大脑擅长解决的。计算机恰好相反，她很难像人一样阅读和理解一段文字，但是进行文字统计并告诉你出现频率最高的词及其出现次数，对计算机而言却是非常容易的：

```
python words.py
Enter file:words.txt
to 16
```

“个人信息分析助理”很快告诉我们，单词“to“在本章前三段中一共出现了16次。

事实上，计算机擅长做人类不擅长做的事，这就是为什么你需要熟练掌握一门与计算机对话的语言。一旦学会这门新语言，你就可以将枯燥的工作指派给你的计算机搭档了，留出更多的时间去做适合你自己的事。在这种合作关系中，你的贡献是才思、直觉力和创造力。

1.1 创新与动机

这本书不是为专业程序员准备的，专业编程是份非常有前途的工作，可算是物质与精神双丰收。为他人创造有用的、简洁的与智能的程序是一项创新性很强的活动。你的计算机或PDA通常安装了来自许多不同程序员开发的各种软件，每一款软件都想要吸引你的注意力和兴趣。它们尽其所能来满足你的需求，在使用过程中让你获得优质的用户体验。在某些情况下，当你选择了一个软件，这个软件的开发者就会因为你的选择而直接获得收益。

如果将程序看作是程序员的创新性产出，那么下图就是一个更形象的PDA模型：

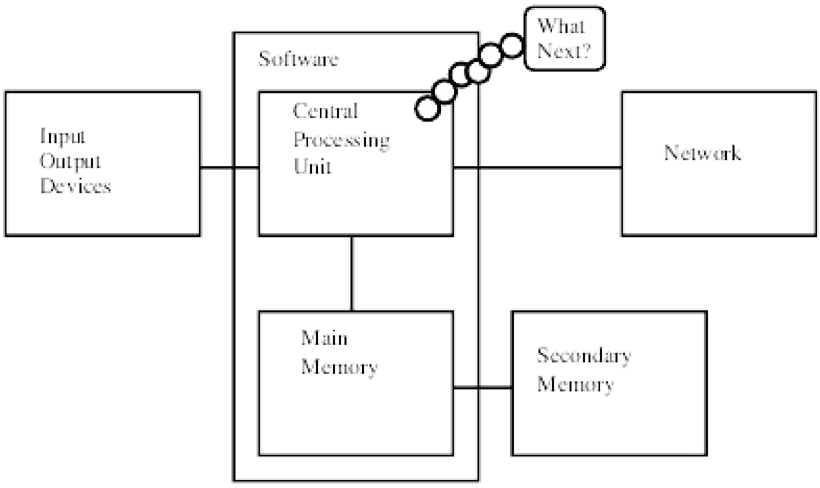


本书的写作初衷不是为了赚钱或者取悦最终用户，而是让我们能更好地处理生活中的数据与信息。开始学编程，你既是程序员，也是你所写程序的最终用户。当你获得了程序员的技能，如果编程让你感到有创新活力的话，到时你的想法也许会发生改变，转向为他人开发程序也说不定。

1.2 计算机硬件架构

学习这种向计算机发指令来开发软件的语言之前，我们需要了解一下计算机的构成。

如果拆开你的计算机或者手机，仔细观察就会发现以下这些组件：



这些组件的一般定义如下：

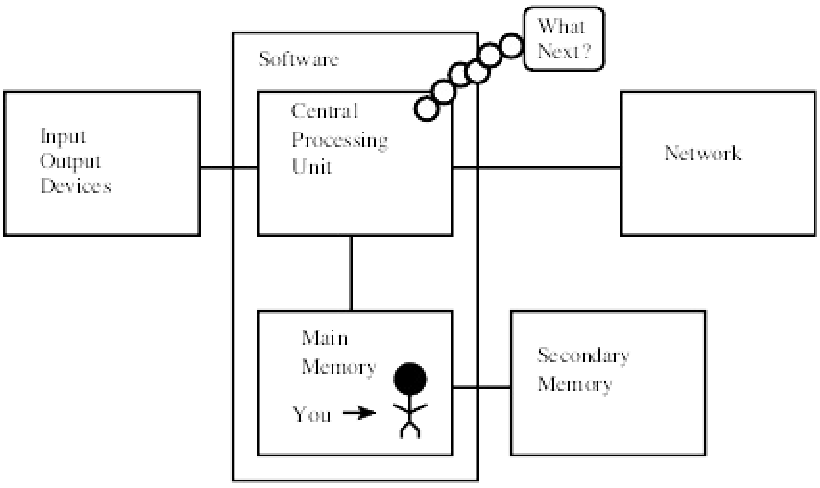
- 中央处理器（Central Processing Unit，CPU）是专门为解决“下一步做什么”而存在的组件。如果计算机处理速度达到3.0 GHz，这就意味着CPU每秒会提问30亿次“下一步做什么？”。你不得不学会如何跟CPU如此快速地交谈与保持同步。
- 主存储器（Main Memory）用来存储CPU即刻需要的信息。主存储器的速度几乎与CPU不相上下。但是，关闭计算机之后主存储器里的信息也就消失了。
- 辅助存储器（Secondary Memory）也是用来存储信息的，但是它比主存储器速度慢很

多。辅助存储器的优点是，它可以在计算机不带电情况下存储信息。常见辅助存储器包括磁盘和闪存。闪存通常用在U盘和便携式音乐播放器上。

- 输入输出设备（Input and Output Devices）包括屏幕、键盘、鼠标、麦克风、扬声器以及触摸板等。这些都是用来与计算机进行交互的设备。
- 如今大多数计算机之间还建立了网络连接，通过网络获取信息。我们可以将网络看成信息存储与检索速度很慢的一个空间，而且不总是那么稳定。从某种意义上讲，网络是速度很慢且并不是那么可靠的辅助存储器。

这些组件的工作原理细节最好还是交给计算机厂商吧。这里只是为了掌握一些术语，在编程时方便提及这些组件。

作为一名程序员，你的工作就是利用并协调这些资源来解决问题和分析数据。作为程序员，你主要与CPU打交道，告诉它下一步做什么。有时，你要告诉CPU调用主存储器、辅助存储器、网络或输入输出设备。



你需要成为回答CPU“下一步做什么”的人。但把你压缩到5毫米高，塞入计算机，让你每秒发出3亿次命令，想必这样会很不舒服。所以，你必须提前写好你的指令。我们把这些存储下来的指令称为程序，编写指令并进行调试的活动称之为编程。

1.3 理解编程

在本书其他章节中，我们尝试把你培养成长为一名理解编程艺术并具备一定编程能力的人。最后，你会成为一个程序员，也许不是专业的。但至少你掌握了如何看待数据（信息）分析问题，并开发出解决问题的程序。

从某种意义上来说，程序员的养成需要两种技能：

- 首先，需要掌握编程语言（Python）本身—— 熟悉词汇和语法。能够准确地拼写这门新语言中的单词，并且掌握如何使用这门新语言正确地“造句”。
- 其次，学会讲故事。在写故事的过程中，通过文字和句式的组合，向读者传达思想。编故事的艺术与能力通过写作与反馈得以提高。在编程中，程序即故事，待解决的问题即

传达的想法。

当掌握一种编程语言（如Python）之后，你会发现学习其他编程语言，如JavaScript或者C++，就会容易许多。虽然新的编程语言拥有很多不同的词汇和语法，但你已经学会解决问题的技能，所有编程语言本质上都是相通的。

Python的词汇和句式上手很快，但要能写出一些连贯的程序来解决一个全新的问题，尚需时日磨练。讲授编程就像讲授写作一样。先对程序进行阅读和解释，然后编写简单的程序，接着逐步编写更复杂的程序。当达到一定水平，你就形成了自己的编程风格，自然而然地去应对问题，通过编写程序解决它。一旦修炼到这个程度，编程就变成一个愉悦且富有创造力的过程了。

我们从Python程序的词汇和结构讲起。第一次阅读时，一定要耐心学习那些简单的例子。

1.4 词汇与句子

与人类语言不同的是，Python的词汇数量实际上相当少。我们称这些"词汇"为"保留字"，它们是Python中具有特殊意义的词汇。对于Python来说，程序中出现的这些词汇，它们有且仅有一个含义。等下你在编程时，你自己定义的词汇称为变量。变量命名非常自由，但有一点，你不能使用Python的保留字作为变量名。

从某种意义上讲，我们训练一只狗时会使用一些特殊的词汇，比如"坐下"、"停下"和"拿来"。跟狗说话时不用这些保留字的话，它们就会傻傻地看着你，直到你对它说出保留字。举例来说，"我希望更多的人通过散步来促进健康。"，而大多数狗听到的可能是，"吧啦吧啦散步吧啦吧啦。"这是因为在狗的语言中"散步"是保留字。很多人可能觉得人类和猫之间的语言没有保留字[1]。

Python的保留字如下：

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

就这么多词汇。Python比狗训练有素多了。当你说"try"，Python会毫无差错地执行try。

后续章节会介绍这些保留字及它们的适用场合。现在，我们只关注怎么与Python对话（就像人跟狗说话）。教Python说话是件有意思的事情，把想要说的话用单引号括起来就可以了。

```
print 'Hello world!'
```

这就是我们写出的第一个语法正确的Python语句。以保留字print开头，后面跟一个文本字符串，用单引号括起来。

1.5 与Python对话

我们已经掌握了Python的一个词汇与一个简单语句，接下来需要了解如何与Python对话，测试我们的新语言技能。

与Python对话之前，必须先在计算机上安装Python软件，学会如何启动Python。本章包含许多细节，建议查看http://www.pythonlearn.com，网站上有Python在Mac和Windows系统上配置和启动的详细说明和视频演示。当打开终端或者命令行窗口，输入python，Python解析器会以交互模式启动，如下所示：

```
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

>>>提示符表示Python解析器在询问，“你希望我下一步做什么？”。Python已经准备好与你对话。你需要掌握的是怎样说Python语言，发起一个对话。

举个例子，你对Python语言最简单的词汇或句子一无所知，想要使用宇航员的标准用语（喊话）。宇航员在一个遥远的星球登陆，试着和这个星球的居民用以下语句对话:

```
>>> I come in peace, please take me to your leader
  File "<stdin>", line 1
    I come in peace, please take me to your leader
         ^
SyntaxError: invalid syntax
>>>
```

事情进展好像并不顺利。除非你反应迅速，否则这个星球的居民可能会拿长矛刺向你，向你吐口水，然后把你放在火上烤，当成晚饭吃掉。

幸运的是，旅行时你带了这本书，及时翻到了这一页，再试一次：

```
>>> print 'Hello world!'
Hello world!
```

这次看起来效果不错，试着与他们继续对话：

```
>>> print 'You must be the legendary god that comes from the sky'
You must be the legendary god that comes from the sky
>>> print 'We have been waiting for you for a long time'
We have been waiting for you for a long time
>>> print 'Our legend says you will be very tasty with mustard'
Our legend says you will be very tasty with mustard
>>> print 'We will have a feast tonight unless you say
  File "<stdin>", line 1
    print 'We will have a feast tonight unless you say
                                                     ^
SyntaxError: EOL while scanning string literal
>>>
```

此时，你应该意识到，Python虽然非常复杂与强大，但在语法上非常挑剔，并不那么智能。对话中必须使用正确的语法。

在某种意义上，当你使用别人写的程序时，Python就在你和其他程序员之间充当中间人。Python是程序编写者将对话进行下去的一种方式。在阅读完短短几章之后，你将成为Python程序员中的一员，与你的程序使用者进行对话。

结束与Python解析器的第一次谈话之前，你可能要知道如何正确地与这个星球的居民说“再见”：

```
>>> good-bye
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'good' is not defined

>>> if you don't mind, I need to leave
  File "<stdin>", line 1
    if you don't mind, I need to leave
             ^
SyntaxError: invalid syntax

>>> quit()
```

你会发现前两个错误提示是不同的。由于**if**是保留字，Python看到保留字会认为我们想说些什么，但句子的语法是错的。

跟Python说“再见”的正确方法是，在交互模式的提示符>>>后输入quit()。猜出这个命令这可能会花费一些时间，所以手头备本书可能会派上用场。

1.6 术语：解释器与编译器

Python是一种高级语言，旨在较为方便地让人类进行读写，让计算进行读取与处理。其他高级语言包括：Java、C++、PHP、Ruby、Basic、Perl以及JavaScript等。CPU里的硬件并不能理解任何一种高级语言。

CPU能理解的语言称之为机器语言。机器语言非常简单，坦白讲，编写起来非常无聊。它全部由0和1组成：

```
0101000111010010010101010000001111
1110011000001110101001010101101101
...
```

虽然机器语言表面看起来很简单，只有0和1，但它的语法比Python复杂得多。所以，很少有程序员用机器语言编程。相反，借助各种翻译器，程序员可以编写像Python或JavaScript这样的高级语言，这些翻译器会将程序转换成机器语言，再交由CPU执行。

因为机器语言依附于计算机硬件，所以不能在不同类型硬件之间移植。使用高级语言编写的程序可以在不同的计算机之间移植，通过在另一台计算机上使用不同的编译器，或者重新编译代码，生成一个适合这台计算机的机器语言版本。

编程语言的翻译器大体可分为两类：(1)解释器 与 (2) 编译器。

解释器读取程序员所写程序的源代码，解析源代码并实时解释指令。Python是一种解释器。当交互式执行Python时，输入一行Python语句，Python就会立即处理它，并做好准备让我们输入下一条语句。

Python语句中有一些地方会告诉Python，你想要Python记住等下会用到的一些数据。这时就需要为数据挑选一个名称来记住它，这样之后就可以通过这个名称来获取对应的数据了。我们使用变量（variable）来代表存储的数据。

```
>>> x = 6
>>> print x
6
>>> y = x * 7
>>> print y
42
>>>
```

在这个例子中，我们让Python记住数值6，并将6赋值给变量x，以便后续使用。为了确认Python已经记住这个数值，使用print命令打印变量x的值。接下来，我们让Python获取变量x的值并乘以7，然后将结果赋给新变量y。最后，打印出当变量y的当前值。

一次输入一行命令，Python将其视为一个语句序列，后面的语句可以获取前面语句的数据。四句组成的段落以一种有逻辑的和有意义的顺序编写，这就是我们写出的第一个简单的多行程序。

如上所示，解释器的本质是进行交互式对话。编译器需要将整个程序放在一个文件中，将高层次的源代码翻译成低层次的机器语言，然后编译器将生成的机器语言放到一个文件中以便后续执行。

如果你使用Windows系统，这些可执行的机器语言程序通常带有“.exe”或“.dll”后缀，分别代表这是“可执行的”和“动态可加载库”。在Linux和Mac中没有这样的后缀来明确表示文件是否是可执行的。

如果在文本编辑器中打开一个可执行文件，满眼望去完全看不懂：

```
^?ELF^A^A^A^@^@^@^@^@^@^@^@^@^B^@^C^@^A^@^@^@\xa0\x82
^D^H4^@^@^@\x90^]^@^@^@^@^@^@4^@ ^@^G^@(^@$^@!^@^F^@
^@^@4^@^@^@4\x80^D^H4\x80^D^H\xe0^@^@^@\xe0^@^@^@^E
^@^@^@^D^@^@^@^C^@^@^@^T^A^@^@^T\x81^D^H^T\x81^D^H^S
^@^@^@^S^@^@^@^D^@^@^@^A^@^@^@^A\^D^HQVhT\x83^D^H\xe8
....
```

机器语言的读写并不容易，好在借助解释器和编译器，能够使用Python或C这样的高级语言编写程序。

通过对解释器与编译器的讨论，你应该对Python解释器本身有了一些了解。那它又是用什么语言写的？是用汇编语言写的吗?当我们输入“python”，究竟发生了什么?

Python的解释器是用C语言编写的。你可以访问http://www.python.org网站，查看Python解释器的源代码，如有你有意愿改造这些源代码也是可以的。Python本身就是一个程序，它被编译成机器代码。当你（或计硬件供应商）在计算机上安装了Python，实际是上将一份编译好的Python程序的机器代码拷贝到你的计算机系统。在Windows中，Python可执行的机器代码很可能位于以下文件夹中:

```
C:\Python27\python.exe
```

要成为一名Python程序员，你知道这些还远远不够。但在一开始，花一些时间解释这些细节问题，还是值得的。

1.7 编写一个程序

体验Python功能的最好方式是在Python解析器中输入命令，但不建议采用这种方式来解决复杂的问题。

编程时，我们在文本编辑器里把Python指令写到一个文件里，这个文件称为脚本。一般而言，Python脚本以.py命名结尾。

为执行脚本，你必须告诉Python解释器脚本文件的名称。在Unix或Windows命令窗口中，你可以输入以下代码来执行hello.py脚本文件：

```
csev$ cat hello.py
print 'Hello world!'
csev$ python hello.py
Hello world!
csev$
```

“csev$”是操作系统提示符，“cat hello.py”是查看“hello.py”文件的内容，其中包含了字符串打印的一行Python程序。

我们调用Python解释器，告诉它从“hello.py”文件中读取源代码，而不是用命令行交互式执行Python代码。

你会发现，没有必要在Python程序文件末尾加上**quit()**。Python读取源代码文件时，到达文件末尾它会自己停止。

1.8 什么是程序？

程序的基本定义是，完成特定任务的一组Python语句序列。最简单的hello.py脚本也是一个程序，不过只是一行代码的程序罢了，没多大作用，但是从严格的定义上来说，它是一个Python程序。

考虑一个可以被程序解决的问题，然后用程序来解决它，这可能是理解程序的最简单方式。

假设，你想对Facebook上的发帖进行社会计算方面的研究，可能感兴趣的是发帖中最常用的词汇是什么。你可以打印出这些发帖，然后通读文本，从中寻找最常见的词，但这需要很长时间而且很容易出错。通过编写Python程序来快速且准确地处理这个任务，这样会比较明智，周末就可以做些其他有趣的事了。

举例来说，阅读以下内容，这是关于一个小丑和一辆车的文本，找出出现次数最多的单词，并统计它出现的次数。

```
the clown ran after the car and the car ran into the tent
and the tent fell down on the clown and the car
```

想象一下，要阅读数百万行文本来完成这个统计任务会是怎样的情形。坦率地说，学习Python，编写一个Python程序来统计的话，要比人工扫描单词快得多。

一个好消息，我用一个简单程序解决了在文本文件中找到最常见单词的这个问题。我已经编写好并测试了这个程序。现在，我把它交给你使用，这样就可以节省你的一些时间。

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
words = text.split()
counts = dict()

for word in words:
   counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word,count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print bigword, bigcount
```

你甚至不需要知道Python就可以使用这个程序。你需要通过本书第10章来完全理解所用到的Python编程技术。你现在是最终用户，只需使用这个程序，你就会惊叹于它的聪明，感叹如何让你摆脱繁重的手工扫描。你只需输入代码，保存成words.py文件并执行它，或者也可以从http://www.pythonlearn.com/code/下载源码。

这个示例充分体现了Python以及这门语言在你（最终用户）与我（程序员）之间扮演的中间人角色。在安装了Python的用户计算机上，Python作为一种通用语言，让我们可以交换有用的指令序列（即程序）。因此，我们并不是跟Python交流，而是通过Python与其他人交流。

1.9 程序的架构

后续章节会详细介绍深Python的词汇、句子结构、段落结构与篇章结构。我们将学习Python的强大功能，以及如何将这些功能组合起来创建有用的程序。

程序的构造包含一些低层次的概念模式。这些构造模式不仅仅针对Python程序，而是每一种编程语言从机器语言到高级语言的通用组成部分。

输入： 从“外部世界”获取数据，可以从文件中读取数据，或者从某种传感器，比如麦克风或GPS获取数据。在我们最初的程序中，输入是用户通过键盘输入数据。

输出：将程序的结果显示在屏幕上，或保存在一个文件，或写入一个设备，如扬声器可以播放音乐或朗读文字。

顺序执行：按照脚本中的语句顺序，一句接一句执行。

条件执行：根据特定条件执行或者跳过特定语句序列。

重复执行：重复执行一些语句，通常也会存在一些变化。

重用：编写好一组指令，为它们命名，之后在程序中根据需要可以再次使用这些指令。

这些听起来好像很容易，事实上并不是那么简单。这就好比，走路很简单吧，“把一只脚放在另一只脚前面”。编程的“艺术”就是不断创造和组合这些基本元素，创造对用户有用的东西。

除了“重用”模式之外，上面的词频统计程序几乎用到了上面提及的所有模式。

1.10 导致出错的原因

在前面与Python的对话中已经看到，我们编写Python代码必须非常精确，很小的偏差和错误都会导致Python放弃执行程序。

初学者通常认为，Python不能容忍犯错，给人留下低劣、可恨与残忍的印象。虽然Python看上去与其他人一样，但它能理解初学者，只是把这种“抱怨”转化为督促，让我们更好地编写程序。不要再说Python老是折磨我们了。

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>> primt 'Hello world'
  File "<stdin>", line 1
    primt 'Hello world'
                      ^
SyntaxError: invalid syntax
>>> I hate you Python!
  File "<stdin>", line 1
    I hate you Python!
         ^
SyntaxError: invalid syntax
>>> if you come out of there, I would teach you a lesson
  File "<stdin>", line 1
    if you come out of there, I would teach you a lesson
              ^
SyntaxError: invalid syntax
>>>
```

与Python争论没什么好处。它只是一个工具，没有情感，它很高兴随时准备为你服务。它的错误信息看起来很苛刻，但这些信息帮助提供线索。Python看到了你输入的内容，但不明白你输入的是什么。

Python更像一只狗，无条件爱你，只能理解很少的关键词，用它那甜美的表情(>>>)看着你，等待你输入一些它能理解的东西。当Python说道：“SyntaxError: invalid syntax”，它只是摇着尾巴说：“你似乎说了什么，但我不太明白，请你继续跟我说话(>>>)。”

当程序变得越来越复杂，你会遇到以下三种类型的错误：

语法错误：

这是你遇到的第一种错误，很容易解决。语法错误意味着，你违反了Python的“语法”规则。Python会尽可能指出一行语句中它所困惑、不能理解的地方和字符。语法错误唯一棘手的是，有时候程序中需要修改的错误可能位于Python所指出困惑的位置前方。所以，Python指出的语法错误位置只是你排查问题的起点。

逻辑错误：逻辑错误是程序没有语法错误前提下，语句顺序或语句关系存在错误。逻辑错误的一个形象举例是，“打开水瓶喝口水，把它放到书包里，走到图书馆，然后再把水瓶盖上。”

语义错误：语义错误是程序的语法完美且逻辑正确，但无法达到预期。也就是说，程序完全正确，但它不能做到你想要它做的事。一个简单的例子，如果你给人指路怎么去一家餐馆，如此说：“...当你走到有加油站的十字路口时，向左转，继续走一英里，你的左手边有一座红色建筑，餐馆就在那儿。”过了很长时间，你的朋友打来电话，他们正在一个农场，围着一个谷仓转圈，并没有看到餐馆的标志。”然后，你问：“你们在加油站左转还是右转了？”他们说：“完全是按你指示的方向走，我还写到纸上，在加油站左转，继续走一英里”。然后，你说：“非常抱歉，虽然我的方向指示在语法上没错，但其中悲催地包含了一个很小的、没有被发现的语义错误。”

对于这三种错误，Python会尽最大努力按照你的要求准确地去执行。

1.11 学习之旅

当循序渐进阅读本书时，如果初次遇到某些概念不能很好理解，不要担心。小时候学说话时，头几年只要能发出一些可爱的喃喃之音，这不是什么问题。花6个月的时间，从能说简单的词汇发展到能表达简单的句子，再用5到6年的时间，从句子上升到段落，这样的发展过程是正常的。再过几年就能依据自身兴趣，独立写出一篇完整的作文了。

我们希望你能以更快地速度学习Python，所以有些内容在接下来的几章中会多次涉及到。学习一门新语言需要花时间去吸收和理解，才能做到运用自如。全景是由一块块小的片段拼接起来的。我们会一再提及一些主题，尝试让你看清全景，这可能会导致一些混乱。由于书的体例是线性的，如果你参加一门课，推进方式是线性的，但不要受此约束。前后来回翻阅，进行一些略读。在没有充分理解细节之前，跳过有难度的内容，这有助于更好的理解编程之道。通过回顾之前的内容，或者重做之前做过的练习，你可能会感收获颇多，其中这些内容可能在你一开始接触时觉得有些费解的。

学习第一门编程语言时，通常会有一些值得欢呼雀跃的时刻。这就像用斧凿精心雕琢，一块岩石最终在你手中变成一尊美丽的雕塑。

如果有些事看起来特别困难，通宵熬夜耗着是没有意义的。休息一下，打个盹，吃点零食，向某人（可能是你的狗）倾诉下你当下遇到的问题，然后，以全新的眼光回过头来再看这个问题。我保证，一旦你从本书中学会了编程知识，再回头看，你会发现编程是非常简单和优雅的行为，只是需要花一些时间去吸收罢了。

1.12 术语

臭虫：程序中的错误。

中央处理单元：所有计算机的心脏。我们编写的软件都由它来执行，也称为“CPU”或者“处理器”。

编译：把高级语言编写的程序翻译成低级语言，为后续执行做好准备。

交互模式：在提示符后输入命令和表达式，这是Python解释器的一种使用方法。

解释：采用一次翻译一行的方式来执行高级语言编写的程序。

低级语言：一种旨在便于计算机执行的编程语言，也称为“机器码”或“汇编语言”。

机器代码： 最接近硬件的编程语言，可直接由中央处理单元（CPU）执行。

主存储器：存储程序和数据。关闭电源后主存储器的信息会丢失。

解析：检查程序和分析语法结构。

可移植性：程序的一个属性，即程序可在不同类型的计算机上运行。

print语句: 能让Python解释器在屏幕上显示数据的指令。

问题解决: 提出一个问题，找到解决方法并形成解决方案的过程。

程序：实现特定计算的一组指令集。

提示：程序显示一个消息，等待用户的输入。

辅助存储器：存储程序和数据，电源关闭后数据不会丢失。辅助存储器的速度通常比主存储器慢。辅助存储器包括磁盘驱动器、U盘中的闪存等。

语义：程序本身的含义。

语义错误：程序的一种错误。程序并未按照程序员意愿做事。

源代码：程序的高级语言代码。

1.13 练习

习题**1.1** 计算机的辅助存储器的功能是什么？

a） 执行程序的所有计算和逻辑

b） 在互联网上检索网页

c） 长期存储信息，甚至重启之后信息不会丢失

d） 接收用户的输入

习题**1.2** 什么是程序？

习题**1.3** 编译器和解释器有什么区别？

习题**1.4** 下面哪个选项包含了“机器代码”？

a）解释器

b）键盘

c）Python源文件

d）word文档

习题**1.5** 找出下面代码的错误：

```
>>> primt 'Hello world!'
  File "<stdin>", line 1
    primt 'Hello world!'
                       ^
SyntaxError: invalid syntax
>>>
```

习题**1.6** 执行以下Python语句，变量“X”存于何处？

```
x = 123
```

a）中央处理器

b）主存储器

c）辅助存储器

d）输入设备

e）输出设备

习题**1.7** 以下程序会输出什么？

```
x = 43
x = x + 1
print x
```

a）43

b）44

c）x＋1

d）出错，因为x = x + 1在数学上讲不通

习题**1.8** 以人作类比，解释以下事物：（1）中央处理单元，（2）主存储器，（3）辅助存储器，（4）输入设备和（5）输出设备。例如，“计算机的中央处理单元相当人体哪个部位”？

习题**1.9** 如何解决“语法错误”？

> 1. http://xkcd.com/231/ ↩

第2章 变量、表达式与语句

2.1 值与类型

值是程序的基本组成要素，如一个字母或一个数字。目前为止，我们接触到的值有1、2和“Hello,World!”。

这些值属于不同的类型：2是整数，’Hello,World!’是字符串（包含一串字母）。你（或者解释器）可以根据有无引号来判别是否是字符串。

print语句也可以打印整数。输入python命令启动解释器。

```
python
>>> print 4
4
```

如果不确定一个值属于哪种类型，解释器会告诉你。

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

显而易见，字符串属于str类型，整数属于int类型。带小数点的数字使用浮点（floating-point）格式表示，称为float类型。

```
>>> type(3.2)
<type 'float'>
```

那么，像'17'和'3.2'这种属于哪种类型呢？看起来像数字，但它们和字符串一样被放在单引号里面。

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

它们是字符串。

输入较大的数字时，可能会在每三个数字之间加一个逗号，例如，1,000,000。在Python中这不是一个整数，但这样表示是合法的。

```
>>> print 1,000,000
1 0 0
```

这不是我们想要的！Python解释器会把1,000,000当做一个逗号分隔的数字序列，它会把三部分依次打印出来，中间用空格分开。

这是我们遇到的第一个语义错误例子：代码运行不会输出任何错误信息，但是它并没有做“正确的事“。

2.2 变量

编程语言最强大的功能之一体现在对变量的操作能力。变量是一个值的引用名称。

赋值语句用来创建新变量并对其赋值：

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

这个例子列举了三种不同类型的赋值语句。第一条语句将字符串赋值给变量message；第二条语句将整数17赋值给变量n，第三条语句将π的近似值赋值给变量pi。

使用打印语句输出这些变量的值。

```
>>> print n
17
>>> print pi
3.14159265359
```

变量的类型就是它所指向的值的类型。

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

2.3 变量名与关键字

程序员通常会选择有意义的变量名，通过变量名较为直观地了解变量的用途。

变量名不限长度，可以同时包含字母和数字，但是必须以字母开头。使用大写字母也是合法的，但使用小写字母会更好（随后会解释原因）。

下划线可以出现在变量名中。它经常用在含有多个词的变量名中，例如，my_name和airspeed_of_unladen_swallow。变量名可以采用下划线开头，但一般要避免这样命名，除非是编写供他人使用的Python库代码。

如果使用不合法的变量名，你就会得到一个语法错误：

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones不是合法的变量名，它不是以字母开头的。more@也是不合法的，它包含了一个不合法的字符@。变量名class错在哪呢？

原因在于，class是Python的保留关键字。Python解释器使用保留关键字来识别程序的结构，因此，保留关键字不能用于变量名。

Python的31个保留关键字如下所示[1]：

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

你可以在手边存留一份。如果解释器指出了一个变量名不合法，而你又不知道为什么，那么检查一下变量名是否在这个列表里面。

2.4 语句

语句是Python解释器能够执行的代码单元。我们已经见到过两个语句：print和assignment。

在交互模式中输入一条语句，解释器就会执行它并打印出结果（如果有结果的话）。

程序举例如下：

```
print 1
x = 2
print x
```

得到以下输出结果：

```
1
2
```

其中，赋值语句没有输出结果。

2.5 运算符和运算对象

运算符是表示运算的一类特殊符号，例如，加法与乘法。运算符操作的值称为运算对象。

+、-、*、/和**五个运算符分别代表加、减、乘、除和次方的运算，请看如下示例：

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

除运算的结果可能不是你所期待的：

```
>>> minute = 59
>>> minute/60
0
```

minute的值是59，在传统的算术中，59除以60是0.98222，而不是0。出现这种差异的原因在于，Python执行的是地板除法（floor division）[2]。

当两个运算对象都是整数时，那么结果也是整数。地板除法默认去掉小数部分，因此上面的运算结果是0。

如果两个运算对象都是浮点数，那么Python会执行浮点除法，结果就是浮点数：

```
>>> minute/60.0
0.98333333333333328
```

2.6 表达式

表达式是值、变量和运算符的组合。值本身可以是一个表达式，变量亦如此。下面都是合法的表达式（假设变量x已被赋值）：

```
17
x
x + 17
```

如果在交互模式中输入一个表达式，解释器就会执行它并把结果打印出来。

```
>>> 1 + 1
2
```

然而，在一个程序中，表达式本身并不能做任何事情！这是初学者容易混淆的一点。

习题**2.1** 在Python解释器中输入下面的语句并查看结果：

```
5
x = 5
x + 1
```

2.7 运算顺序

当一个表达式中出现多个运算符时，运算顺序按照一定规则进行。对于数学运算符来说，Python遵照数学运算习惯，“先括号再次方接着乘除后加减”（PEMDAS）会帮助你记住运算顺序。

- 括号拥有最高运算优先级，让表达式按特定顺序执行运算。括号内的表达式优先进行运算，例如，2 (3-1) 等于4，(1+1)**(5-2) 等于8。有时候，括号即便没有改变运算结果，但可以阅读起来更加方便，例如，(minute 100) / 60。
- 幂运算的级别仅次于括号，例如，2**1+1 等于3，而不是4，3*1**3等于3，而不是 27。
- 乘法和除法具有相同的优先级，高于加法和减法。例如，2*3 -1等于5，而不是 4，6+4/2等于8，而不是5。
- 相同优先级的运算符按从左到右的顺序依次运算。例如，5-3-1等于1，而不是3。先计算5-3得到的2，然后再减1。

当不能确定运算顺序时，通常使用括号来确保既定的运算顺序。

2.8 模运算

模的运算对象是整数，得到的是第一个整数与第二个整数相除的余数。在Python中，模运算符用百分号（%）表示，语法与其他运算符一致：

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

7被3所除的商是2，余数是1。

模运算非常实用。举例来说，你可以检验一个数是否能被另一个数整除，如果x%y的结果是0，那么x能被y整除。

另外，模运算也可以提取出一个数字最右边的数位。举例来说，x%10可以提取出一个数最右边的一位数字（以10为基数）。同理，x%100可以提取出一个数最右边的两位数字。

2.9 字符串运算符

加号的运算对象是字符串，它会把字符串首尾相连。这里不是数学意义上的加号。例如：

```
>>> first = 10
>>> second = 15
>>> print first+second
25
>>> first = '100'
>>> second = '150'
>>> print first + second
100150
```

这个程序的输出结果是100150。

2.10 请求用户输入

有时候我们希望获取用户通过键盘输入的值。Python提供了一个输入函数raw_input，用来获取键盘输入[3]。当调用这个函数时，程序会暂停运行，等待用户的输入。当用户按下回车键时，程序就恢复运行，raw_input函数以字符串形式返回用户输入的值。

```
>>> input = raw_input()
Some silly stuff
>>> print input
Some silly stuff
```

从用户处获取输入之前，最好打印一条提示语句，告诉用户可以输入些什么。在程序暂停等待用户输入之前，你可以在raw_input中插入一个字符串来提示用户。

```
>>> name = raw_input('What is your name?\n')
What is your name?
Chuck
>>> print name
Chuck
```

提示语结尾的 \n 表示新开辟一行，它是一个用于换行的特殊字符。这样一来，用户输入的位置出现在提示语句的下面。

如果希望用户输入一个整数，你可以尝试int()函数，将返回的值转换成整数型：

```
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
>>> int(speed) + 5
22
```

但是，如果用户输入的不是数字组成的字符串，那么就会报错：

```
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
```

随后会介绍如何处理这类错误。

2.11 注释

当程序变得越来越复杂，阅读难度也随之增大。正规的程序代码很密集，经常会遇到看不懂这段代码是做什么的，或者为什么要这样写。

为解决这个问题，在程序代码中加入自然语言说明，解释这段代码的作用，这会是一个不错的主意。这些说明称为注释，它们以#号开头：

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

上面这种情况，注释本身占一行。你也可以把它加到一行代码的末尾：

```
percentage = (minute * 100) / 60     # percentage of an hour
```

从#号开始到这一行的最后，编译时都会被忽略掉，它们不会对程序产生任何影响。

对代码不显著的特征进行说明时，注释非常有用。注释会帮助代码阅读者搞清楚这段代码的作用。注释用于代码为什么这样写的解释，同样有用。

下面这行注释明显是多余的，没什么作用：

```
v = 5     # assign 5 to v
```

而下面的这行注释则包含了有用的信息：

```
v = 5     # velocity in meters/second.
```

清晰易懂的变量名能够减少注释的使用，但是变量名如果太长，就会使复杂的表达式变得更加难懂，所以需要权衡利弊。

2.12 选择易记的变量名

只要遵循变量命名的简单规则，避免使用保留字，变量的命名还是有很多种选择的。编程入门阶段，你在阅读别人的程序和编写自己的程序时，对变量的命名可能会感到困惑。例如，下面三个程序实质上是一样的，但是阅读和理解起来差别很大。

```
a = 35.0
b = 12.50
c = a * b
print c

hours = 35.0
rate = 12.50
pay = hours * rate
print pay

x1q3z9ahd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ahd * x1q3z9afd
print x1q3p9afd
```

Python解释器对这三个程序的处理方式完全一样，但是对于读者理解起来差异很大。读者能够快速看懂的是第二个程序，这是因为程序员选择了能够代表变量取值含义的变量名。

这种变量命名法称为“助记变量命名法”。助记[4]的意思就是帮助记忆。选择易于记忆的变量名，有助于我们记住当初创建这个变量是为了做什么。

这看起来不错，使用助记变量命名法是一个好主意，但可能也会对初学者解析与理解代码上产生负面影响。由于初学者可能还没有记全Python的31个保留字，如果变量名中包含太多描述性的词语，精心命名的变量看上去会像是Python语言的一部分，对初学者理解上造成干扰。

下面两行简单的Python代码实现了循环。循环将在第5章介绍，这里尝试猜猜这两行代码的含义：

```
for word in words:
    print word
```

执行上面的代码会发生什么？for、word、in等这几个词中哪些是保留字，哪些是变量名？Python能理解单词的基本含义吗？初学者很难分辨出代码中哪些部分必须照抄示例中的，而哪些部分是可以自主选择的。

下面的代码和上面的代码实质上是一样的：

```
for slice in pizza:
    print slice
```

通过观察两段代码，初学者可以容易地分辨哪些是Python保留字，哪些是程序员自己定义的变量名。显而易见，Python不能理解pizza和slice之间的差别，实际上就是一个披萨可以切成很多块。

如果程序要读取数据并在数据中查找单词，pizza和slice是不容易记住的变量名。选择它们作为变量名会偏离程序的本意。

过不了多久，你会熟悉最常用的保留字并在程序中注意到它们：

for word **in** words: **print** word

这段Python代码中for 、in 、print和:加粗显示，程序员自己定义的变量名不加粗。很多文本编辑器支持Python语法，它们会使用不同的颜色来标记保留字，让你能方便地区分保留字与变量名。熟悉一段时间后，你就会很快地区分哪些是保留字，哪些是变量名。

2.13 调试

最容易出现的语法错误是不合法的变量名，例如，class和yield是保留字，或者包含不合法字符的odd~job和US$。

如果变量名中存在空格，Python会认为它是没有运算符的两个运算对象：

```
>>> bad name = 5
SyntaxError: invalid syntax
```

语法错误的提示信息提供不了什么帮助。最常见的错误信息是SyntaxError: invalid syntax和SyntaxError: invalid token，这两条都没有提供什么有价值的信息。

最常遇到的运行错误是“use before def;”，这个错误的意思是使用了一个还没有赋值的变量。这种情况容易出现在变量名拼写错误的情况：

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

变量名区分大小写，例如，LaTeX和latex是不一样的。

最容易出现的语义错误是运算顺序。举例来说，\(\dfrac{1}{2π} \)可能会写成这样：

```
>>> 1.0 / 2.0 * pi
```

这个语句首先进行除法运算，得到的结果是π/2，显然跟\(\dfrac{1}{2π} \)不是一回事！Python并不明白这个表达式是什么意思。这种情况下虽然不会报错，但计算结果是错误的。

2.14 术语

赋值：给变量赋予一个值的语句。

连接：将两个运算对象首尾相接。

注释：程序里面包含的信息，旨在帮助其他程序员（或者查看源代码的人）理解程序，不会对程序的执行产生任何影响。

求值：对表达式执行运算，得到一个值。

表达式：变量、运算符和值的组合，表示一个结果值。

浮点：带有小数部分的数值。

地板除法：截掉两数相除所得结果的小数部分的一种除法运算。

整数型：代表整数类型。

关键字：Python解释器用来解析程序的保留字。变量命名不可使用保留字，例如，if、def与while等。

助记法： 一种辅助记忆的方法。通常使用易记的变量名来帮助我们理解变量本身包含的内容。

模运算：用百分号（%）表示，求两数相除的余数。

运算对象：运算符操作的对象。

运算符：能够进行简单运算的一类特殊符号，例如，加法、乘法和字符串连接。

运算优先级：一组运算规则，用来规定多个运算符和运算对象的表达式中运算执行的次序。

语句：包含指令和行动的一段代码。目前为止，我们见到了赋值语句和打印语句。

字符串：由字符序列组成的数据类型。

类型：表示一类值。目前为止，我们见到了整数（int），浮点数（float）和字符串（str）。

值：数据的基本单位，例如，程序中可以操作的数字或字符串。

变量：一个值的引用名称。

2.15 练习

习题**2.2** 使用raw_input编写一个程序，提示用户输入姓名，然后打印出欢迎语。

```
Enter your name: Chuck
Hello Chuck
```

习题**2.3** 编写一个程序，让用户输入工时和时薪，然后计算出工资总额。

```
Enter Hours: 35
Enter Rate: 2.75
Pay: 96.25
```

暂时不用担心结果是否精确到小数点后两位。如果需要精确到小数点后两位，可以使用Python内置的round 函数。

习题**2.4** 执行下面的赋值语句：

```
width = 17
height = 12.0
```

写出以下表达式的值及值的类型。

1. width/2

2. width/2.0
3. height/3
4. 1 + 2 * 5

使用Python解释器检查你的结果是否正确。

习题2.5 编写一个程序，让用户输入摄氏温度，然后将其转化成华氏温度，最后将转化后的温度值打印出来。

> [1]. Python 3.0中，exec不再是保留关键字，nonlocal是新的保留关键字。 ↩

> [2]. Python 3.0中，做除法的结果是浮点数。新的运算符//做整数除法。 ↩

> [3]. Python 3.0中，这个函数称为input。 ↩

> [4]. "助记"的有关详细介绍，请参见http://en.wikipedia.org/wiki/Mnemonic ↩

第3章 条件执行

3.1 布尔表达式

布尔表达式是具有真或假状态的一种表达式。==运算符用来比较两个运算对象，若两者相等返回True，否则返回False：

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True和False是布尔类型的两个取值，它们不是字符串：

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
```

==运算符是比较运算符中的一个，其他的比较运算符如下：

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
x is y               # x is the same as y
x is not y           # x is not the same as y
```

虽然你可能很熟悉这些运算符，但要注意这些Python符号并不等同于数学符号。一个常见的错误是用单等号（=），而没有用双等号（==）。请记住，=是赋值运算符，==是比较运算符。不存在=<或=>这样的运算符。

3.2 逻辑运算符

逻辑运算符包括and（与）、or（或）与not（非）三种。这些运算符的语义与它们的英文含义相似。例如，

```
x > 0 and x < 10
```

若x大于0且小于10，则为真。

若 `n%2 == 0 or n%3 == 0` 其中有一个条件为真，即这个数字能被2或3整除，则为真。

not运算符是表示否定的布尔表达式。若 `x > y` 为假，即x小于或者等于y，那么 `not (x > y)` 为真。

严格讲，逻辑运算符的运算对象应该是布尔表达式，但在Python中并不是很严格。任何非零数字都可看作是“真”。

```
>>> 17 and True
True
```

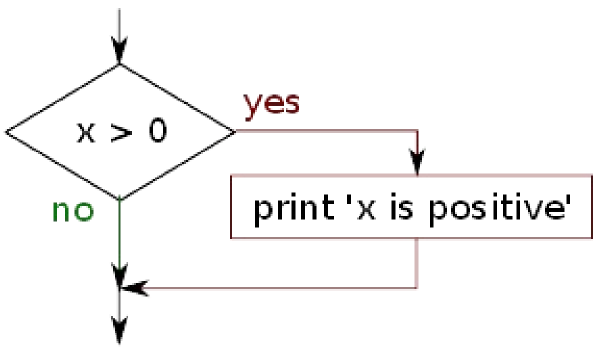
这种灵活性的存在是有用的，但也会产生一些微妙的困惑。除非你清楚自己在做什么，否则不要乱用。

3.3 条件执行

为了编写出有用的程序，几乎总是需要根据条件，修改程序相应的行为。条件语句赋予我们这种能力。最简单的条件形式是if语句：

```
if x > 0 :
    print 'x is positive'
```

if语句后的布尔表达式称为条件。if语句的末尾用冒号 (:)，if语句之后的语句要缩进。



若逻辑条件为真，那么缩进的语句得以执行。若逻辑条件为假，那么缩进的语句就会被跳过。

if语句与for循环[1]的函数定义具有相同的结构。语句由一个以冒号（:）结束的标题行和随后的一个代码缩进区域构成。由于包含多行，这样的语句被称为复合语句。

语句块内没有语句的数量限制，但至少要有一行。有时候，空白的语句块也是有用的，通常是为还未写的代码预留空间。在这种情况下，使用pass语句，表示什么也不做。

```
if x < 0 :
    pass          # need to handle negative values!
```

如果在Python解释器里输入一个if语句，提示符会从三个“>”变成三个“.”，表明正处在一个语句块内，如下所示：

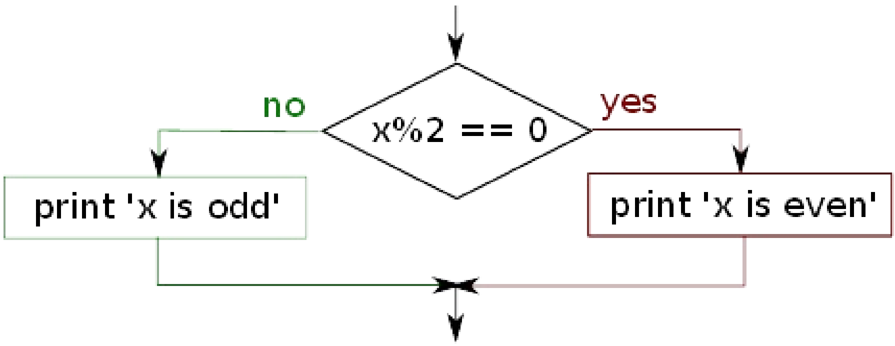
```
>>> x = 3
>>> if x < 10:
...     print 'Small'
...
Small
>>>
```

3.4 分支执行

if语句的第二种形式是选择执行。语句中存在两种可能，条件决定了执行哪一种。语法如下：

```
if x%2 == 0 :
    print 'x is even'
else :
    print 'x is odd'
```

若x除以2的余数为0，则x是偶数，程序就执行第一个print语句，显示“x is even”。若余数不为0，则else之后的第二个print语句得以执行。

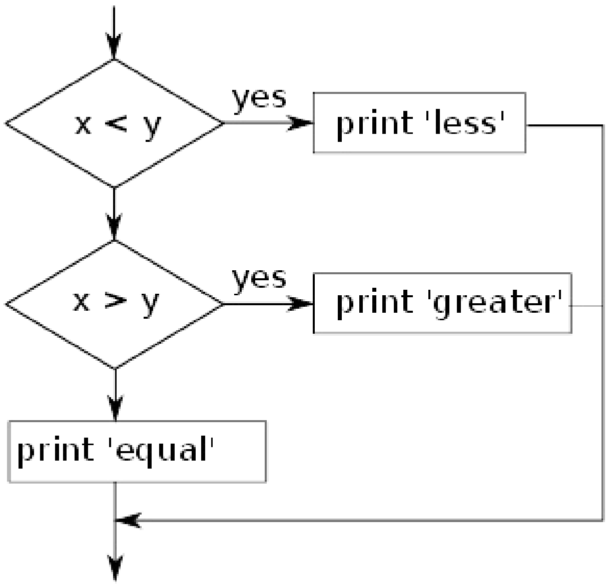


由于条件必须为真或假，所以两条备选语句中总有一条会被执行。这些备选语句称为执行流程中的分支。

3.5 链式条件

有时候存在两种以上的可能，那么就需要两个以上的分支。链式条件可以处理这种情况。

elif是“else if”的缩写，表示又有一个分支将被执行。



elif语句没有数量限制。如果还有一个else子句，那它必须是最后一个，但是else子句不是必须的。

```
if choice == 'a':
    print 'Bad guess'
elif choice == 'b':
    print 'Good guess'
elif choice == 'c':
    print 'Close, but not correct'
```

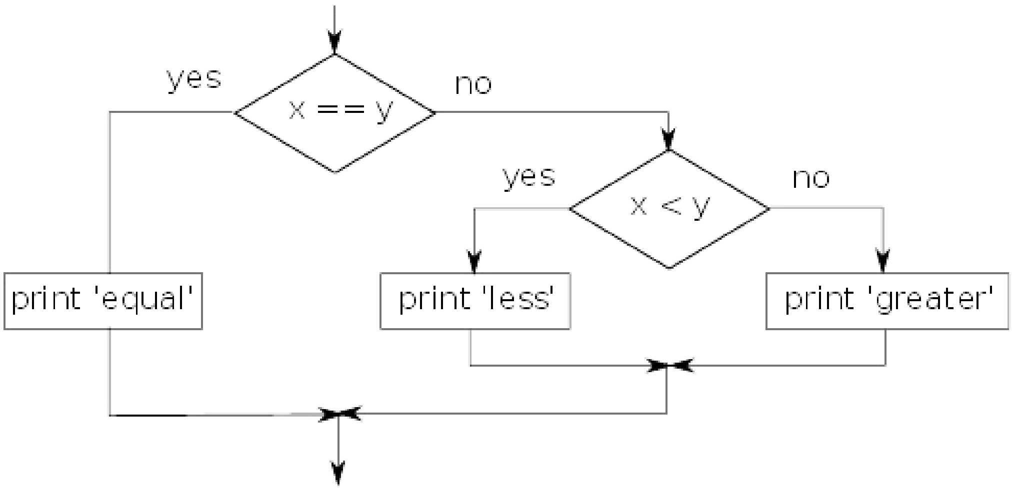
依次检查每个条件。如果第一个为假，就检查第二个，以此检查下去。如果其中一个条件为真，则执行相应的分支，至此语句结束。即使不止一个条件为真，也只执行第一个为真的分支。

3.6 嵌套条件

一个条件语句也可以嵌套到另一个条件语句中。我们可以写出三分条件的例子：

```
if x == y:
    print 'x and y are equal'
else:
    if x < y:
        print 'x is less than y'
    else:
        print 'x is greater than y'
```

外面的条件语句包含了两个分支。第一条分支包含了一个简单语句。第二条分支包含了另一个由两个分支组成的if语句。这两个分支都是简单语句，虽然这两个分支包含条件语句，但它们都是简单语句。



虽然缩进让语句结构变得清晰，但是嵌套条件还是很难快速地阅读。一般情况下，还是尽量避免使用。

逻辑运算符通常提供一种方法来简化嵌套条件语句。例如，使用单个条件改写以下代码：

```
if 0 < x:
    if x < 10:
        print 'x is a positive single-digit number.'
```

当两个条件都满足时，print语句才会执行。使用and运算符也能达到相同的效果：

```
if 0 < x and x < 10:
    print 'x is a positive single-digit number.'
```

3.7 try与except异常捕获

之前我们看到过一段代码，使用raw_input和int函数读取和解析用户输入的整数。由此带来的潜在危险是：

```
>>> speed = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int()
>>>
```

在Python解释器中执行这些语句，提示"值错误"，然后会新起一行，等待下一条语句的输入。

如果把这段代码放在Python脚本文件中，当错误发生时，脚本的执行会立即停止在这一行，并返回错误消息，之后的语句不会被执行。

```
inp = raw_input('Enter Fahrenheit Temperature:')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print cel
```

如果我们执行这段代码，输入一个无效的值，它会停止执行，并返回一个不友好的错误信息：

```
python fahren.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren.py
Enter Fahrenheit Temperature:fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: invalid literal for float(): fred
```

Python内置了“try/except”条件执行结构，用来解决意料之中和意料之外的错误。你知道程序可能存在问题，希望在错误发生时增加一些语句，这时try 和 except就派上用场了。如果没有出错，这些额外的语句（except语句块）就会被忽略掉。

你可以把Python 的try 和 except功能看作是程序的保险单。

温度转换程序重新编写如下：

```
inp = raw_input('Enter Fahrenheit Temperature:')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print cel
except:
    print 'Please enter a number'
```

Python首先执行try语句块。如果一切顺利，它就会跳过except语句块。如果在try语句块里发生意外，Python就会跳出try语句块，执行except语句块。

```
python fahren2.py
Enter Fahrenheit Temperature:72
22.2222222222

python fahren2.py
Enter Fahrenheit Temperature:fred
Please enter a number
```

用try语句处理异常的行为称为异常捕获。这个示例中except子句打印了一条错误提示信息。一般来说，捕捉到异常，就是给你一个机会去解决它，或者是再试一次，至少程序能正常结束。

3.8 逻辑表达式的短路评估

当Python处理诸如 `x > = 2 and (x / y)> 2` 这样的逻辑表达式时，从左至右进行判断。根据and的含义，如果x小于2，则表达式 `x>=2` 为假，那么整个表达式即为假，不管后面的 `（x / y）> 2` 的判断是真或假。

当Python发现逻辑表达式剩余部分的判断没有意义了，它就停止对剩余部分的判断。在已知逻辑表达式整体结果的情况下停止判断，这称为短路评估。

这看起来是一个细节，短路行为带来了一种灵活的处理方式，称为守护者模式。如下是Python解释器中的一段代码：

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

第三次计算出错了。由于y等于0，Python计算 `（x/y）` 时出现运行错误。但是，第二次计算没有出错。由于表达式第一部分 `x > = 2` 判断为假，所以根据短路规则， `(x / y)` 根本没有被执行，所以没有报错。

我们可以在错误发生之前，策略性地放置一个守护评估，代码如下：

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

第一个逻辑表达式中 `x > = 2` 为假，所以在and之前判断就停止了。第二个逻辑表达式中 `x > = 2` 为真，但 `y != 0` 为假，所以 `(x/y)` 没有机会得到判断。

第三个逻辑表达式中 `y != 0` 位于 `(x/y)` 之后，所以这个表达式的判断失败了。

第二个表达式中 `y != 0` 作为一个守护，确保y不等于0时只执行(x/y)。

3.9 调试

当错误发生时，Python的追踪（Traceback）显示很多信息，但有可能消息过多，特别是栈中有很多栈帧的情况。其中最有用的部分是：

- 错误的类型是什么
- 错误发生的位置

语法错误通常很容易找到，但也存在一些陷阱。空格错误非常棘手，由于空格和制表符是不可见的，常常会被忽视。

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
SyntaxError: invalid syntax
```

这个例子中问题出在第二行缩进只用了一个空格。但是错误消息指向了y，这其实是一种误导。一般情况下，错误信息会显示问题在何处被发现，但是实际错误可能会在所显示的代码之前，有时候还会在前一行。

运行时错误也是如此。假设用分贝计算信噪比，公式是\(SNR~{db} = 10 log~{10}(\frac{P~{signal}}{P~{noise}})\)，Python代码如下:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
print decibels
```

但是当你执行这段代码，就会收到如下错误信息[2]：

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
OverflowError: math range error
```

错误信息显示第5行出错，但那一行没有错误。为了找到真正的错误，将ratio值打印出来可能会有用。错误原来是ratio为0，问题出现在第4行，原因是两数相除采用的是地板除法（只取整数）。解决方法是用浮点数表示信号功率和噪声功率。

一般情况下，错误信息会告诉你问题出现在哪，但这个出错位置并不准确。

3.10 术语

主体：复合语句中的一组语句。

布尔表达式：取值只有真（True）或假（False）其中之一的表达式。

分支：条件语句中可供选择的一组语句。

链式条件：带有多个可选分支的条件语句。

比较运算符：对运算对象进行比较的一种运算符，包括==、!=、 >、 <、 >=,和 <=。

条件语句：根据某些条件来控制程序执行顺序的语句。

条件：条件语句中的布尔表达式，用来决定执行哪一个分支。

复合语句：含有头部和主体的一组语句。代码头部以冒号（:）结尾。代码主体相对于代码头部进行缩进。

守护模式：通过额外的比较来构造逻辑表达式，充分利用短路行为优势。

逻辑运算符：组合布尔表达式的运算符，包括and、or和not。

嵌套条件：一个条件语句作为另一个条件语句的分支。

追踪：正在执行的函数列表，当出现异常时，在屏幕上显示出来。

短路：在判断逻辑表达式的过程中，如果Python已经知道了最终结果，则会停止，不会对剩余的表达式进行判断。

3.11 练习

习题**3.1** 重写薪水计算公式，如果员工工作时间超过40小时，按平常薪水的1.5倍支付。

```
Enter Hours: 45
Enter Rate: 10
Pay: 475.0
```

习题**3.2** 运用try和except重写支付程序，让程序可以正常处理非数字输入的情况，如果是非数字输入，打印消息并退出程序。以下是程序的两种执行结果：

```
Enter Hours: 20
Enter Rate: nine
Error, please enter numeric input

Enter Hours: forty
Error, please enter numeric input
```

习题**3.3** 编写一个程序，提示分数在0.0和1.0之间。如果分数超出这个范围则打印出错误。如果分数在0.0和1.0之间，使用下面的表格打印分数：

```
Score   Grade
>= 0.9     A
>= 0.8     B
>= 0.7     C
>= 0.6     D
< 0.6    F

Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

重复运行这个程序，测试不同的输入值。

> 1. 第4章介绍函数，第5章介绍循环。↩

> 2. Python3.0中，不再给出错误消息。即使是整数对象，除法运算符执行的也是浮点除法。↩

第4章 函数

4.1 函数调用

编程中的函数是执行计算的一个命名语句序列。当定义函数时，需要指定函数名和语句顺序。定义好之后，通过函数名就可以调用函数。之前我们见到过一个函数示例：

```
>>> type(32)
<type 'int'>
```

这个函数的名称是type 。括号里面的表达式称为函数的参数。参数可以是一个值或变量，作为函数的输入。type函数的作用是显示参数的类型。

一般而言，函数获取参数，然后返回结果。这个结果称为返回值。

4.2 内置函数

Python提供了很多重要的内置函数，不需要我们预先定义就可以使用。Python开发者定义了解决常见问题的函数集，内置在Python中，供我们使用。

max函数和**min**函数分别求得一个列表中的最大值和最小值：

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

max函数告诉我们，这个字符串中最大的字符是（“w”），min函数告诉我们这个字符串中最小的字符是空格符。

另一个常用的内置函数是len函数，它能返回参数的长度。如果len函数的参数是一个字符串，那么它返回的是这个字符串的字符数。

```
>>> len('Hello world')
11
>>>
```

这些函数不仅可用于字符串，还能操作其他数据类型，后续章节会涉及到。

你可以将内置函数类比为保留字，举例来说，不要使用“max”作为变量名。

4.3 类型转换函数

Python提供了将一种类型转换成另一种类型的内置函数。在允许的情况下，int函数能把任何其他类型的值转化成整数型，如果不能转换就会报错：

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int函数能将浮点型转换成整数型，但是它不进行四舍五入，只是直接去掉小数部分：

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float函数能把整数型和字符串型转换成浮点型：

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

str函数能把传入的参数转换成字符串型：

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.4 随机数

既定输入的情况下，大部分的计算机程序每次都会产生相同的输出，也就是说这是板上钉钉的事情。一般而言，确定性是好的，我们希望同一个算法产生相同的结果。然而，有些时候我们希望计算机具备不确定性，比如游戏就是一个典型例子，当然还有很多其他例子。

让一个程序完全处于不确定状态，这很难做到。但是，有一些方法可以让它看起来具有不确定性。一种方法是利用算法产生伪随机数。伪随机数并不是真正的随机数，原因在于它们是利用确定的算法计算出来的。单看这些数字并不会发现它们和真正的随机数有什么不同。

随机数模块提供了伪随机数生成的函数，以下简称“random”。

random函数返回一个介于0.0-1.0的随机浮点数（包括0.0，但不包括1.0）。每次调用时，你会得到一长串的数字。举例来说，执行下面的循环语句:

```
import random

for i in range(10):
    x = random.random()
    print x
```

程序生成了10个介于0.0-1.0之间但不包括1.0的随机数。

```
0.301927091705
0.513787075867
0.319470430881
0.285145917252
0.839069045123
0.322027080731
0.550722110248
0.366591677812
0.396981483964
0.838116437404
```

习题**4.1** 运行以下程序并观察得到的结果，不妨多运行几次看看。

random函数只是众多随机数函数中的一个。randint函数传入两个整数参数，并返回介于这两个参数之间的整数（包括这两个参数在内）。

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

为取得一组值中的随机元素，可以使用下面的程序：

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

random模块还提供了从连续分布函数中提取随机值的函数，包括高斯函数、指数函数、伽玛函数等。

4.5 数学函数

Python的math模块提供了很多常见的数学函数。在使用之前要导入相应的包：

```
>>> import math
```

这条语句创建了一个名为math的模块对象。如果将这个模块对象打印出来，就会得到关于它的一些信息：

```
>>> print math
<module 'math' from '/usr/lib/python2.5/lib-dynload/math.so'>
```

这个模块对象包括了模块中定义的函数和变量。为了使用其中的函数，你必须指定模块名和函数名，用圆点隔开，也就是英文的句号。这种格式叫做点标记（dot notation）。

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

第一个例子是计算以10为底的信噪比的对数。math模块还提供了log函数用来求以e为底的对数。

第二个例子调用了正弦函数。变量名字给出了提示，除了正弦函数，还有余弦，正切等其他三函数都采用弧度作为参数。将度数转换成弧度的运算是除以360，然后乘以2π：

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

math.pi表达式从math模块中取得变量pi的值。这个变量是π的一个近似值，保留了小数点后15位数。

运用你的平面几何知识，与2的平方根再除以2的结果进行比较，检查程序输出是否正确。

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

4.6 定义新函数

目前为止，我们只是使用了Python的内置函数，除此之外也可以自定义新函数。通过定义函数名和一组语句序列来定义一个新函数，然后在执行时调用这个函数。一旦定义了一个函数，程序中可以重复使用。

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'
```

def是用来定义函数的保留关键字。这个函数的名称是print_lyrics。函数命名与变量命名的规则基本上是一样的。字母、数字以及一些符号是合法的，但是函数名的第一个字符不能是数字。不能使用保留关键字命名函数，也要避免函数名和变量名相同。

函数名后面的空括号表明这个函数没有指定参数。接下来，我们会定义有参数的函数。

函数定义的第一行叫做函数头，剩余的部分叫做函数体。函数头必须以冒号结束，函数体必须缩进。按照惯例，一般缩进4个空格。函数体可以包括任意数量的语句。

print语句要输出的字符串放在双引号里面。单引号和双引号作用是一样的。除了在字符串中也有单引号的冲突情况之外，大多数人会使用单引号。

如果你在Python交互模式中定义函数，解释器会打印省略号（...）表明函数定义还没有结束。

```
>>> def print_lyrics():
...     print "I'm a lumberjack, and I'm okay."
...     print 'I sleep all night and I work all day.'
...
```

函数定义的结束方式是输入一个空行（当然，这在程序文件中不是必须的）。

定义函数的同时也会创建一个和函数名相同的变量名。

```
>>> print print_lyrics
<function print_lyrics at 0xb7e99e9c>
>>> print type(print_lyrics)
<type 'function'>
```

print_lyrics的值是一个函数对象，它的类型是函数。

调用自定义函数与调用内置函数的语法是一样的：

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

函数定义好之后，就可以在其他函数中使用。例如，为了重复调用之前的函数，你可以写一个repeat_lyrics函数：

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

然后调用这个函数：

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

现实中歌曲可不是这样填词的。

4.7 定义与用法

将前面的代码片段组合在一起，完整的程序如下：

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

这个程序包含了两个自定义函数：print_lyrics 和 repeat_lyrics。函数定义会像其他语句一样执行，但它的作用仅仅是新建函数。函数体里面的代码并没有被执行，直到被调用时才会执行，函数定义是没有输出的。

正如所期望的，你必须在执行函数之前定义这个函数。换句话说，函数在第一次被调用之前就应该定义好。

习题**4.2** 将上面程序的最后一行调到最开始，这样函数调用就出现在函数定义之前。运行程序并观察出错信息。

习题**4.3** 将函数调回底部，将函数print_lyrics定义放在函数repeat_lyrics后面，这样会产生什么结果？

4.8 执行流程

为了保证函数定义是在第一次使用之前，你就必须知道语句的执行顺序，这称为执行流程。

程序一般从第一条语句开始执行，从上到下依次顺序执行。

函数定义并不改变程序的执行顺序，但要记住，函数只有被调用的时才会被执行。

函数调用像是是执行流程中的一个迂回。当遇到要调用的函数时，不会接着去执行下一句，而是跳到函数体，执行完函数体里的语句，然后回到刚才跳出的地方继续执行。

当你掌握了函数之间的调用，这就会变得很简单。在一个函数中，程序可能需要执行另一个函数的语句。还有，在定义新函数时，程序还可能需要执行另一个函数！

程序里跳来跳去，这说明了什么？阅读程序有时候没必要从头看到尾。有时候按照执行流程会让你更好地读懂程序。

4.9 形式参数与实际参数

我们之前见到的内置函数需要传入参数。例如，调用math.sin函数需要传入一个数值作为参数。有些函数不止有一个参数：math.pow函数需要传入两个参数：底数和指数。

在函数内部，如果变量作为参数传入，那么这种参数叫做形式参数（简称形参）。下面的例子是一个自定义函数传入了一个实际参数（简称实参）：

```
def print_twice(bruce):
    print bruce
    print bruce
```

这个函数将一个叫做bruce的实参传递进去。当该函数被调用时，函数会打印两次传入的参数值。

这个函数适用任何类型的值。

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

这种参数传入方式适用于内置函数与自定义函数。任何类型的表达式都可以作为函数print_twice的参数。

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

参数的值在函数被调用之前就已经被计算出来了，因此，上面的例子中表达式'Spam '*4 和math.cos(math.pi)只被计算了一次。

你还可以使用变量作为参数：

```
>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

变量（如michael）作为形参与实参（如bruce）不同。函数的形式参数是什么类型的值都可以被传入，而实参的值是确定的，print_twice函数的返回值就是参数bruce本身。

4.10 带返回值的函数和空函数

我们使用过的有些函数（如math函数）有返回值。由于没有更好的叫法，我称之为有返回值的函数。其他的一些函数比如print_twice，执行任务但不返回值，这种称为空函数。

当调用有返回值的函数时，大部分时候你想要利用返回值，例如，将它赋值给一个变量，也可以作为表达式的一部分：

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

在交互模式中调用函数时，Python会显示结果。

```
>>> math.sqrt(5)
2.2360679774997898
```

在程序文件中，如果不把有返回值的函数的返回值保存在变量里的话，返回值就会丢失。

```
math.sqrt(5)
```

空函数也许会在屏幕上输出什么或者有其它的用途，但其本身没有返回值。如果你试着将结果赋值给一个变量，就会得到一个特殊的值None。

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

None值与字符串'None'是不一样的。它是一种特殊的值并且拥有自己的类型。

```
>>> print type(None)
<type 'NoneType'>
```

使用return语句从函数中返回结果。例如，创建一个简单的函数addtwo，将两个数相加并且返回一个值。

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print x
```

在文件中执行程序时，print语句会打印出8。addtwo函数被调用时，传入3和5这两个参数。带有形参a和b的函数，分别是3和5。这个函数将两个数相加，然后将结果赋值给局部变量added，最后使用return语句将计算结果赋值给x并打印出来。

4.11 为什么需要函数？

如果不太明白为什么需要花费精力把程序分成多个函数，请看如下理由：

- 定义新函数可以让你有机会为一组语句命名，使得程序易于阅读、理解和调试。

- 函数可以消除冗余代码，为程序瘦身。如果想修改函数，只需要改变一处即可。
- 将长程序分割成多个函数，逐一调试，最后再整合在一起。
- 精心设计的函数对很多程序而言都会有用。一旦创建并调试后，你就可以重复使用它。

本书其余部分会使用函数定义来解释一个概念。创建并正确使用函数的技巧之一是捕捉到一个想法，例如，“找出列表中的最小值”。随后我们会介绍min函数的代码，它将列表的值作为参数输入，然后返回其中的最小值。

4.12 调试

制表符和空格通常不可见，导致不易调试，因此最好选择一个能够自动缩进的文本编辑器。

另外，运行程序之前不要忘记保存。有一些开发环境可以自动保存，而有一些不行。未保存的情况下运行的程序和文本编辑器的程序不一样。

如果你重复运行相同错误的程序，那么会调试很久。

确保运行的代码是文本编辑器中你所看到的代码。如果不确定，你可以在程序开头加入 `print 'hello'` 这样的代码再运行。如果没有看到hello输出，那么就没有正确运行程序。

4.13 术语

算法：解决一类问题的一般过程。

实际参数：函数被调用时传入的值。函数中这个值会被赋予相应的参数。

函数体：函数定义中的一组语句序列。

组合：一个表达式作为另一个表达式的一部分，或者一条语句作为另一条语句的一部分。

确定性：在既定输入下，程序每次运行都一样并返回相同的结果。

点标记：在另一个模块中调用函数时，需要指定模块名和函数名，用句号隔开。

执行流程：程序运行时的语句执行顺序。

带返回值的函数：能够返回一个值的函数。

函数：执行一些有用操作的命名语句序列。函数不一定有参数，也不一定返回结果。

函数调用：执行函数的语句，包括函数名和参数列表。

函数定义：创建新函数的语句，需要指定名称、参数和执行语句。

函数对象：通过函数定义创建的值。函数名就是参引函数对象的变量。

函数头部：函数定义的第一行。

import语句：读取模块文件并创建模块对象的语句。

模块对象：import语句创建的一个值，用以提供模块中定义的数据和代码。

形式参数：函数中用来参引传入参数的变量名。

伪随机数：看起来像随机数的一串数值，不过这是由确定的程序产生的。

返回值：函数的结果。如果一个函数调用作为表达式来使用，那么返回值就是这个表达式的值。

空函数：没有返回值的函数。

4.14 练习

习题**4.4** Python中"def"保留关键字的作用是什么？

a) 它是一个俚语，"下面的代码太酷了"

b) 它表示函数的开始

c) 它表示接下来代码的缩进部分会被存储

d) b和c都正确

e) 以上都不正确

习题**4.5** 下面的Python语句会打印出什么？

```
def fred():
   print "Zap"

def jane():
   print "ABC"

jane()
fred()
jane()
```

a) Zap ABC jane fred jane

b) Zap ABC Zap

c) ABC Zap jane

d) ABC Zap ABC

e) Zap Zap Zap

习题**4.6** 重写之前的工资计算程序，加班时间的工资按原工资的150%计算。创建一个computepay函数，包含两个参数hours和rate。

Enter Hours: 45

Enter Rate: 10

Pay: 475.0

习题**4.7** 重写之前的分数计算程序，使用computegrade函数，score作为参数，返回一个评分等级的字符串。

```
Score   Grade
> 0.9     A
> 0.8     B
> 0.7     C
> 0.6     D
<= 0.6    F


Program Execution:

Enter score: 0.95
A

Enter score: perfect
Bad score

Enter score: 10.0
Bad score

Enter score: 0.75
C

Enter score: 0.5
F
```

重复运行该程序，每次输入不同的值来测试输出结果。

第5章 迭代

5.1 更新变量

赋值语句的常见模式是对变量进行更新，变量的新值依赖于旧值。

```
x = x+1
```

这条语句的作用是：得到x的当前值，加一，然后将相加结果作为新值赋予x。

如果更新一个不存在的变量，那么会出错。这是因为Python在给x赋值之前，首先计算等号右边：

```
>>> x = x+1
NameError: name 'x' is not defined
```

在更新变量之前，必须初始化，通常使用简单的赋值语句：

```
>>> x = 0
>>> x = x+1
```

通过加一操作更新变量称为增量更新，减一操作称为减量更新。

5.2 while语句

计算机经常用于执行一些重复性的任务。对计算机来说，执行相同或相似的任务而不出错，这很简单，但人就做不好。迭代很常见，Python提供了一些功能语句，使这类任务变得更加简单。

Python的一种迭代形式是while语句。下面是一个简单的例子，从5开始倒数，然后打印出“Blastoff!”。

```
n = 5
while n > 0:
    print n
    n = n-1
print 'Blastoff!'
```

几乎可以像读英文一样，读懂这条while语句。它的作用是：当n大于0时，显示n的值，然后对n减1。当n等于0时，结束while语句，显示“blastoff!”。

严格说， while语句的执行流程如下：

1. 计算条件表达式的值，判断是True或False。
2. 如果为False，结束while语句并执行下一条语句。
3. 如果为True，执行while中的语句体，然后返回步骤1。

此类执行流程称为循环。执行到第三步又返回到顶部。每执行一次循环体，称为一次迭代。上面的循环，我们可以说“它进行了五次迭代”，表示循环体被执行了五次。

循环体会改变一个或多个变量的值。因此当条件不满足时，循环结束。有一种变量在每次循环执行时其值都会变化，并控制循环什么时候结束，这种变量称为迭代变量。如果没有迭代变量，循环就会永远执行下去，导致无限循环。

5.3 无限循环

对于程序员来说，无限循环的有趣实例就是洗发水的说明书，“泡沫，冲洗，重复”。这就是一个无限循环，没有迭代变量来表明什么时候结束这个循环。

上面的倒数例子中，循环的确是结束了。因为n值的个数是有限的，我们可以看到n值随着循环的执行不断减小，最终变为0。有些情况的循环明显是无限的，这是因为它根本就没有迭代变量。

5.4 “无限循环”与break语句

有时候循环运行到一半时，你还没意识到这时候该结束循环了。在这种情况下，你可以写一个无限的循环，然后使用break语句跳出循环。

下面的代码明显是一个无限循环，while语句的逻辑表达式是常量True：

```
n = 10
while True:
    print n,
    n = n - 1
print 'Done!'
```

如果犯了这个错误并且运行这个代码，你会很快学会如何停止一个正在运行的Python进程，或者找到计算机的关机按钮。由于循环顶部的逻辑表达式是常量True，所以循环条件一直都满足，这个程序会一直运行下去，直到计算机没电。

虽然这是一个不正常的无限循环，我们还是可以使用这种模式来建立有用的循环。只要将break语句放进循环体，明确退出条件及行为。当达到结束条件时，就可以结束循环。

举例来说，如果想要用户直到输入done时结束的话，代码可以这样写：

```
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line
print 'Done!'
```

这个循环的条件是True且不会变，因此循环会一直执行下去，直到触发break语句。

每次执行这个循环，它都会提示用户一个尖括号。如果用户输入done，那么break语句就会结束这个循环。否则，这个程序会一直提示用户进行输入，回到顶部继续执行。下面是一个程序运行的结果演示：

```
> hello there
hello there
> finished
finished
> done
Done!
```

while语句的这种写法很常见。你可以在循环中的任何位置检查条件（不仅局限于顶部），并且可以主动定义停止条件（当发生什么就停止），而不是被动等待判断（一直运行直到发生什么）。

5.5 使用continue语句结束迭代

有时在循环的迭代中，你想要结束当前迭代，立刻进行下一轮迭代。在这种情况下，使用continue语句跳入下一轮迭代，无需完成当前迭代的循环体。

下面的循环例子不断打印输入值，直到用户输入“done”才会结束。但是，#号开头的输入不会被打印出来（这有点像Python的注释）。

运行一下这个加入了continue语句的新程序。

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line
print 'Done!'
```

除了#号开头的行，其他所有的行都被打印出来。当continue语句被执行，当前迭代会结束，跳到while语句的开头执行下一轮循环，这样也就跳过了print语句。

5.6 使用for语句定义循环

有时候我们需要要遍历一组东西，例如，单词列表，文件的每一行或是一组数字。遍历一组东西，可以用for语句来构造循环。因为wile语句是简单地进行循环，直到条件变为False，我们称其为无限循环。与之不同的是，for语句是对已知的数据项集合进行循环，因此它的迭代次数取决于数据项的个数。

for循环和while循环的语法相似，下面是一个for语句和循环体代码示例：

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print 'Happy New Year:', friend
print 'Done!'
```

在Python语法中，friends变量是包含三个字符串的列表[1]。for循环遍历整个列表，依次打印每个字符串这三个字符，输出结果如下所示：

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

如果用英语来解释这个for循环，就不如while循环的解释那么直接。你可以把它当做一个朋友名单，那么这段代码的作用是：对friends集合中的每个朋友执行for循环体。

观察这个for循环，for和in是Python的保留关键字，friend和friends是变量。

for friend **in** friends:

    print 'Happy New Year', friend

具体来说，friend是for循环的迭代变量。变量friend的值在每次迭代时都会改变，并控制for循环什么时候结束。这个迭代变量取得friends中存储的三个字符串。

5.7 循环模式

我们经常使用for循环或while循环来遍历列表或文件的内容，还会通过浏览来寻找一组数值中的最大值或最小值。

此类循环的构造方法如下：

- 循环开始之前初始化一个或多个变量。
- 在循环体中对每个数据项进行计算，这样可能会改变循环体中变量的值。
- 循环结束时查看最终变量。

我们会用一组数字来展示这些循环模式的理念和构造方法。

5.7.1 统计与求和循环

举例来说，为了统计一个列表的数据项个数， for循环编写如下：

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print 'Count: ', count
```

循环开始之前将变量count的值设为0，然后用一个for循环来遍历数值列表。迭代变量命名为itervar。虽然我们并不在循环体中使用它，但它控制着循环，让循环体为每一个列表值执行一次。

在循环体中，列表的每个值都会导致count值加一。随着循环的执行，count值就是“当前”所看到的。

循环一旦结束，count值就等于列表中数值的个数。在循环最后，总数“落入我们手中”。通过构造这个循环，在循环结束时我们得到了想要的。

另一个相似的循环求出一组数值的总和：

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print 'Total: ', total
```

在这个循环中，我们用到了迭代变量。不是之前循环中为count变量简单加一，而是在每次循环中加上实际的数字（如3、41、12等）。total变量的作用是求出目前的累计值。在循环开始之前，由于还没有遇到任何值，所以total值是0。循环中会累加，total最终值是所有数字的总和。

随着循环的进行，total累积了列表各项的和。这样的变量有时候被称为累加器（accumulator）。

不管统计循环还是求和循环，在实际使用中都不是很有用。这是因为Python提供了内置函数len()和sum()，分别计算列表元素的个数和列表各项的总和。

5.7.2 最大值与最小值循环

找出列表或者序列中的最大值，构造如下循环：

```
largest = None
print 'Before:', largest
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print 'Loop:', itervar, largest
print 'Largest:', largest
```

程序执行结果如下：

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

largest变量是“目前我们看到的最大值”。在循环开始之前，将largest设为常量None。None是一个特殊的常量，表示变量为“空”。

在循环开始之前，我们没有遇到任何值，所以largest值为None。当循环开始执行，如果largest为None，则将第一个值作为目前看到的最大值。可以看到，第一轮迭代中itervar的值是3，largest的值是None，所以立即将largest的值变为3。

第一次迭代之后，最大值就不是None了。复合逻辑表达式的第二部分设置了触发器，检查itervar值是否大于largest值。如果当前值大于largest值，就会将更大的新值赋予largest。从程序输出可以看出，largest值从3变为41，然后变为74。

循环结束后遍历了所有的值，largest值已经是整个列表中的最大值了。

计算最小值的代码仅需要做很小改动：

```
smallest = None
print 'Before:', smallest
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print 'Loop:', itervar, smallest
print 'Smallest:', smallest
```

同样的，smallest值在循环前、循环中与循环后都是“目前看到的最小值”。循环结束后，smallest值就是整个列表的最小值。

统计和求和同样有Python内置函数支持，比如，使用max()函数和min()函数可以不用到循环代码。

下面是Python内置函数min()的简化版代码：

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

在这段代码中，我们移除了所有的print语句，与Python内置函数min()基本等价。

5.8 调试

当程序越写越长，你就会发现需要花费更多的时间来调试。代码越多意味着犯错的机会越大，隐藏的错误也就越多。

相反，试着将问题分为两部分。在程序的中间位置，寻找一处可验证的代码，插入一个print语句（或者其它可验证效果的语句），然后运行程序。

如果中间点检查出错，那么问题肯定出在程序的前半部分。如果中间点检查没错，问题就出在程序的后半部分。

每执行一次这样的检查，你就会把代码范围缩减一半。如果代码少于100行，进行6次之后，就只剩下一两行，理论上至少是这样。

实际上，“程序的中间位置”并不是很明显，也可能没法进行检查。统计行数然后除以2，找到精确的中间位置代码行，这种做法没有意义。一般做法是，考虑程序中容易出现错误的地方，进行检查。选择你认为极有可能会出错的位置前后，设置检查点。

5.9 术语

累加器：循环语句中用来累积结果的变量。

计数器：循环语句中用来计算发生次数的变量。计数器初始化为0，每次需要计数时，增加它的值。

减量：减少变量值的更新。

初始化：为随后会更新的变量赋予初始值的语句。

增量：增加变量值的更新（通常是加一）。

无限循环：终止条件无法达到或者没有终止条件的循环。

迭代：通过递归函数调用或者循环来重复执行一组语句。

5.10 练习

习题**5.1** 编写一个程序，重复读取数据，直到用户输入“done”。一旦输入“done”，打印总和、个数与平均值。如果用户输入的不是数字，使用try和except捕获异常，打印错误信息，然后跳过继续执行循环。

习题**5.2** 编写一个程序，提示用户输入一组数字，输出最大值和最小值，不要求平均值。

> 1. 第8章会详细介绍列表。 ↩

第6章 字符串

6.1 字符串是字符的序列

字符串是若干字符的序列。你可以用方括号运算符逐一访问每个字符：

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

第二条语句从fruit变量中提取索引位置为1的字符，并把它赋予letter变量。

方括号里的表达式称为索引。索引可以指向字符序列中你想要的字符，作用如其名。

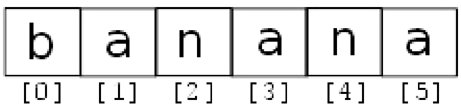
但是这并不是你想要的结果：

```
>>> print letter
a
```

大多数人认为'banana'的第一个字符是b，而不是a。但在Python中，索引是从字符串头部算起的一个偏移量，第一个字母的偏移量为0。

```
>>> letter = fruit[0]
>>> print letter
b
```

因此，b是'banana'的第零个字母，a是第一个字母，n是第二个字母。



索引可以是任何表达式，包括变量与运算符在内。但是，索引的值必须是整数，否则会得到如下错误信息：

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

6.2 使用len函数得到字符串的长度

len是内置函数，返回字符串的字符数：

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

为了得到字符串中的最后一个字母，你可能会这样做：

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

IndexError的错误依据是字符串’banana’没有与索引6对应的字母。既然是从零开始算起，六个字母的编号就是0到5。为了得到最后一个字母，对length值减1。

```
>>> last = fruit[length-1]
>>> print last
a
```

另一种方法是使用负索引，从字符串结尾倒过来计算。表达式fruit[-1]表示最后一个字母，fruit[-2]是倒数第二个字母，以此类推。

6.3 通过循环遍历字符串

按照一次一个字符的方式处理一个字符串需要花费大量的计算。通常从字符串头部开始，逐个选择每个字符，对其进行操作，然后继续下一个，直到结束。这种处理模式称为遍历（traversal）。遍历的一种写法是使用while循环:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index = index + 1
```

这个循环遍历了整个字符串，每行显示一个字母。循环条件是 `index < len(fruit)` ，当index等于字符串长度时，循环条件为假，那么语句体就不会被执行。字符串最后一个字符的索引是 `len(fruit)-1` 。

习题**6.1** 编写一个while循环，从字符串的最后一个字符开始，反向逐一处理，直到字符串的第一个字符，一行显示一个字母。

遍历的另一种写法是用for循环：

```
for char in fruit:
    print char
```

每一次循环中字符串的下一个字符赋值给char变量。继续循环下去，直到没有字符了。

6.4 字符串分割

字符串的一个片段称为切片，字符切片与字符选择的方法相似：

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:12]
Python
```

运算符[n:m] 返回字符串从第n到第m之间的字符，包括第一个字符，但不包括最后一个字符。

如果忽略第一个索引值(冒号之前)，切片就从字符串第一个字符开始计算。如果忽略第二个索引值，切片就计算到最后一个字符：

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

如果第一个索引值大于第二个索引值导致空字符串，只会输出两个引号：

```
>>> fruit = 'banana'
>>> fruit[3:3]
"
```

空字符串不包含字符，其长度为0。除此之外，它和其它字符串没有差别。

习题**6.2** 假设fruit是一个字符串，那么fruit[:]表示什么？

6.5 字符串是不可变的

在赋值语句的左边使用[]运算符，尝试改变字符串中的字符。举例如下：

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

这个例子的对象是字符串，数据项就是你想要赋值的字符。现在，一个对象相当于一个值，等下会修正这个定义。数据项是序列中的一个值。

出错原因在于字符串是不可改变的。这意味着，你不能改变已经存在的字符串。最好的办法是在原字符串基础上新建一个字符串。

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

这个例子将新的首字母与greeting的切片连接在一起。这不会对原先的字符串造成影响。

6.6 循环与统计

下面的程序统计了字母a在字符串中出现的次数：

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print count
```

这个程序演示了另一种计算模式，称为计数器。变量count初始值为0，每当发现一个a，count值就加一。当循环停止时，count就得到了结果，即a出现的总次数。

习题**6.3** 定义一个count函数并封装这段代码，对其进行通用化改造，能够接收字符串和字母作为参数。

6.7 in运算符

单词in是一个布尔运算符，对两个字符串进行比较，如果第一个字符串是第二个字符串的子串，则返回True。

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

6.8 字符串比较

比较运算符适用于字符串。如何判断两个字符串等价：

```
if word == 'banana':
    print  'All right, bananas.'
```

其它比较运算符适用于字母排序：

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

Python不能像人一样，区分大写字母和小写字母。所有的大写字母都在小写字母之前，因此结果如下：

```
Your word, Pineapple, comes before banana.
```

解决这个问题的常见方法是将字符串转换成一种标准格式，例如，在比较之前都转化为小写。

6.9 字符串方法

字符串是一种Python对象。一个对象包括数据（即字符串本身）和方法。这些方法是内置在对象中的有效函数，可以作用于对象的任一实例。

Python有一个dir函数，它可以列出对象所有可用的方法。type函数显示对象的类型，dir函数显示的是对象可用的方法。

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
>>>
```

当dir函数列出这些方法，你就可以用help获取关于这些方法的文档。有关字符串方法比较全面的文档详见http://docs.python.org/library/string.html。

调用方法与调用函数类似，都是传入参数，返回结果，但它们的语法是不同的。调用方法的语法是，使用句点作为分隔，在变量名后面跟上方法名。

例如，upper方法接收一个字符串，返回一个全部是大写字母的新字符串：

这次不使用upper(word)函数，换做word.upper()方法。

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

这种点标记形式指明方法名为upper，将此方法应用于变量word。空的圆括号表示这个方法没有参数。

召唤一个方法称为调用。这个例子中，在word对象上调用了upper方法。

例如，字符串方法find，找到字符串中字符的所在位置：

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

在这个例子中，我们在word对象上调用find方法，将待寻找的字母作为参数。

find方法不仅适用字符，还可以用于寻找子串：

```
>>> word.find('na')
2
```

find方法还可以设置第二个参数，从哪个索引位置开始查找：

```
>>> word.find('na', 3)
4
```

一个常见任务是利用strip方法移除字符串首尾的空白（包括空格、制表符和换行符）。

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

像startswith这样的方法返回的是布尔值。

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
```

你会注意到，startswith方法对大小写敏感，在检查之前，使用lower方法将其全部转换为小写字母。

```
>>> line = 'Please have a nice day'
>>> line.startswith('p')
False
>>> line.lower()
'please have a nice day'
>>> line.lower().startswith('p')
True
```

最后一个例子中调用了lower方法，然后用startswith方法检查小写转换后的字符串是否以字母p开头。只要留意次序，我们就可以在一个表达式上运用多种方法。

习题**6.4** 字符串方法count与之前练习过的函数相似。请访问http://docs.python.org/library/string.html，查看这个方法的文档，编写一个方法调用，统计a在'banana'中出现的次数。

6.10 字符串解析

通常，我们想要在一个字符串中寻找它的子串。如下是一行结构化的字符串：

```
From stephen.marquard@ uct.ac.za Sat Jan  5 09:14:16 2008
```

我们只想抽出电子邮件的第二部分（即uct.ac.za），可以通过find方法和字符串切片来实现。

首先，在字符串中找到@符号的位置。其次，找到@符号之后第一个空格所在的位置。最后，再用字符串切片来提取字符串中我们需要的部分。

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ',atpos)
>>> print sppos
31
>>> host = data[atpos+1:sppos]
>>> print host
uct.ac.za
>>>
```

这里使用的是find方法的一种用法，让我们能指定find方法从何处开始寻找。当切分字符串时，我们提取的字符是“位于@符号之后但不包括空格”的字符。

有关find方法的文档，请访问http://docs.python.org/library/string.html。

6.11 格式操作符

格式操作符%可以构建字符串，使用变量中存储的数据来替代字符串的一部分。对整数而言，%是模运算符。如果第一个操作对象是字符串，那么%就是格式操作符。

第一个操作对象是格式字符串，它包含一个或多个格式化序列，用来指定第二个操作对象的格式。最终处理结果是字符串。

例如，格式序列'%d'表示第二个操作对象会被格式化为整数型（d表示十进制）：

```
>>> camels = 42
>>> '%d' % camels
'42'
```

运行的结果是字符串'42'，不要与整数42搞混了。

格式序列可以出现在字符串中的任意位置，所以你可以在一个语句中嵌入一个值：

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

如果字符串中存在多个格式序列，那么第二个参数必须是元组。每个格式序列与元组的元素依次对应。

下面的例子使用'%d'格式化整数，'%g'格式化浮点数（不要问为什么），'%s'格式化字符串：

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
```

元组中元素的数量必须与字符串中的格式序列数量一致。另外，元素的类型也要与格式序列匹配：

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

第一个例子中元素的数量不够，第二个例子中元素的格式是错的。

格式运算符很强大，但不那么容易上手。如需了解更多，请访问 http://docs.python.org/lib/typesseq-strings.html。

6.12 调试

编程时经常问问自己，“这里可能出现什么样的错误？”或者“我们的用户可能会做怎样疯狂的事情使（看似）完美的程序崩溃？”。这是需要长期培养的编程意识。

例如，第5章迭代中介绍while循环的程序示例如下：

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line

print 'Done!'
```

当用户输入一个空行会发生什么：

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#' :
```

输入空行之前代码运行正常。由于没有第0位字符，我们得到了异常信息反馈。两种方法可以解决这个问题，即使这一行为空，仍然能保证"安全"运行。

一种方法是使用startswith方法，如果字符串为空就返回False。

另一种方法是使用守护模式，通过一条if语句进行控制，保证第二个逻辑表达式只有在字符串中至少有一个字符时进行判断。

```
if len(line) > 0 and line[0] == '#' :
```

6.13 术语

计数器：用来统计的变量。通常初始化为零，然后累增。

空字符串：不包含字符、长度为零的字符串，用两个引号表示。

格式操作符：%操作符对格式字符串和元组进行操作，根据特定格式字符串，对元组元素进行格式化后生成一个字符串。

格式序列：格式字符串中的字符序列，例如，%d，它表示一个值应该如何进行格式化。

格式字符串：使用了格式操作符的字符串，它包含了格式序列。

标记：用来表示条件是否为真的布尔变量。

调用：方法的召唤语句。

不可变：序列的一种属性，序列中的数据项不能被赋值。

索引：用来选择序列中数据项的一个整数值，例如，表示字符串中一个字符的位置。

数据项：序列中的一个值。

方法：与对象相关的函数，使用句点来调用。

对象：变量可以引用的东西。现在，你可以交替使用“对象”或者“值”。

搜索：找到所要找的内容才会停止的一种遍历模式。

序列：一个有序集合，集合中的每个值通过一个整数索引定位。

切片：根据索引区间指定字符串的一部分。

遍历：遍历序列中的数据项，对每个数据项执行类似的操作。

6.14 练习

习题**6.5** 使用以下语句存储一个字符串：

```
str = ’X-DSPAM-Confidence:  0.8475’
```

使用find方法和字符串切片，提取出字符串中冒号后面的部分，然后使用float函数，将提取出来的字符串转换为浮点数。

习题**6.6** 访问http://docs.python.org/lib/string-methods.html，阅读字符串方法的文档。你可能想要拿它们操作一下，以便理解它们的工作原理。字符串方法中的strip和replace特别有用。

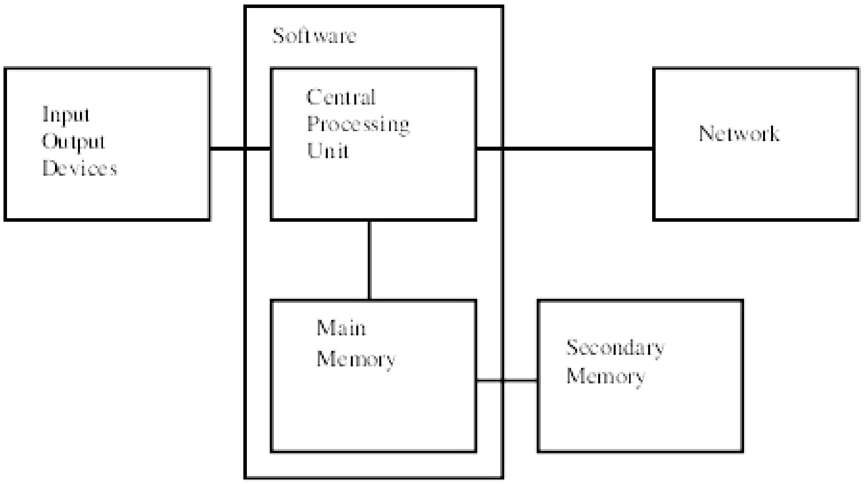
文档中使用的语法可能会使人困惑。例如，在find(sub[, start[, end]])中，方括号表示可选的参数。sub是必需的，start是可选的；如果包含了start，那么end就是可选的。

第7章 文件

7.1 持久性

截止目前，我们已经学习了如何编写程序，通过条件执行、函数和迭代等手段，与中央处理器CPU沟通我们的意愿。我们还学习了如何在主存储器中创建与使用数据结构。CPU和内存是软件工作与运行的地方。它也是所有计算机进行“思考”的大脑。

回顾一下之前讨论过的硬件架构问题，电源关闭后CPU与主存储器（也就是内存）中的数据就会丢失。之前我们学习编写的Python程序仅作为临时的练习。



本章介绍辅助存储器（也就是文件）。电源关闭后辅助存储器里的数据不会丢。借助U盘（闪存），我们可以把程序从一个系统转移到另一个系统。

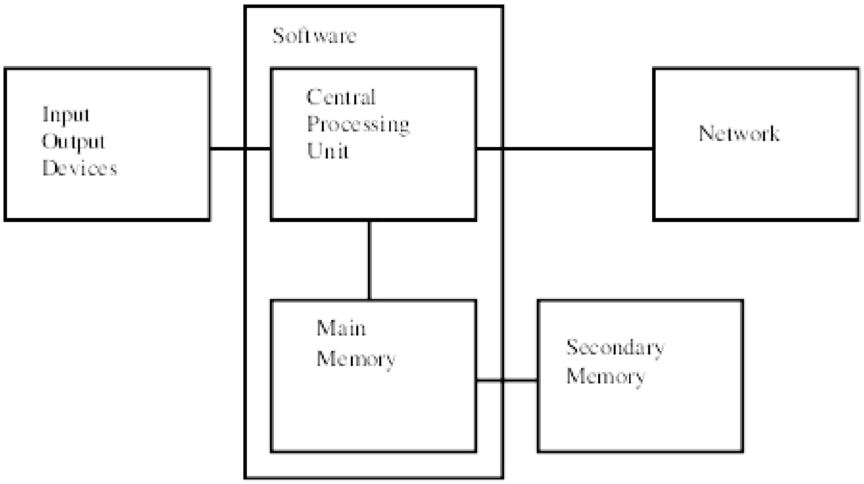
首先，我们学习文本文件的读写。这里的文本文件是指我们在文本编辑器中创建的。然后，我们会了解如何与数据库文件进行交互，例如，二进制文件，通过数据库软件进行读写。

7.2 打开文件

从硬盘上读写文件之前，必须首先打开这个文件。打开文件需要与操作系统进行对话，它知道文件数据的存储位置。打开一个文件时，让操作系统通过文件名找到它，并确定这个文件是否存在。在下面的例子中，我们打开mbox.txt这个文件，它应该存储在你运行的python程序同一个文件夹下。这个示例用来打开mbox.txt文件。这个文件存储在你启动Python时所在的那个文件夹。从http://www.py4inf.com/code/mbox.txt下载这个文件。

```
>>> fhand = open('mbox.txt')
>>> print fhand
<open file 'mbox.txt', mode 'r' at 0x1005088b0>
```

如果文件成功被打开，操作系统会返回一个文件句柄。文件句柄并不存储文件的实际数据，只是一个“操作”，用于读取数据。如果请求的文件存在，你会得到一个句柄并获得读取该文件的相应权限。



如果文件不存在，打开失败，输出追踪错误信息，无法得到访问文件内容的句柄。

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'stuff.txt'
```

稍后，我们会使用try和except更好地处理文件打开时文件不存在的情况。

7.3 文本文件与文本行

文本文件可视为若干文本行的序列，这与Python字符串是字符的序列道理相同。举例来说，以下是一个文本文件示例，记录了在一个开源项目中开发团队成员的邮件活动。

邮件交流的整体文件可以从 http://www.py4inf.com/code/mbox.txt下载。文件的缩减版可以从http://www.py4inf.com/code/mbox-short.txt下载。文件中包含多条邮件，遵守一种标准格式。以“From ”开头的行是每一条邮件第一行，以“From:” 开头的行是邮件的一部分，注意区分。更多信息请访问http://en.wikipedia.org/wiki/Mbox。

将文本文件分解成文本行，“一行的结束”用专门的字符来表示，称为换行符。

Python字符串常量的换行符用\n表示。这看起来像两个字符，事实上它是一个字符。在Python解释器中输入变量stuff，\n出现在字符串中，使用print语句可以看到字符串被换行符分成了两行。

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print stuff
Hello
World!
>>> stuff = 'X\nY'
>>> print stuff
X
Y
>>> len(stuff)
3
```

因为换行符是一个字符，所以字符串“X\nY”的长度是3。

基于以上考虑，逐行读取文件时，我们需要知道每一行的结尾都有一个特殊的隐藏字符，即换行符。

总之，换行符将文本文件分解为若干文本行。

7.4 读取文本行

虽然文件句柄并不包含文件的数据，但它可以方便地构建一个for循环，按行依次读取文件。

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
print 'Line Count:', count

python open.py
Line Count: 132045
```

for循环中文件句柄被当做序列来使用。在这个示例中，for循环只是简单地计算并输出文件的行数。这个循环大致用英语可以翻译成：“每遇到文件中的一行（表示为文件句柄），将count变量值加一”。

open函数如果没能读取整个文件，可能是文件过大，其中包含许多个G的数据。不论文件大小，open语句的打开时间是不变的。最终，for循环读取了文件中的数据。

以for循环这种方式读取文件时，Python根据换行符将文件数据分成若干文本行。Python根据换行符获得一行，每次迭代中将换行符作为一行的最后一个字符。

由于for循环每次只读取一行数据，因此它能高效地读取与统计出大型文件的文本行，无需耗尽内存来存储数据。上面的程序只使用很少的内存就能统计处任意大小文件的文本行，它先读取，然后统计，最后舍弃。

如果文件大小相对于内存容量来说很小，那么就可以把它当做一个字符串，在文件句柄上使用read方法一次性读取进来。

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print len(inp)
94626
>>> print inp[:20]
From stephen.marquar
```

在这个示例中， mbox-short.txt整个文件（总计94626个字符）直接被读入到变量inp中。我们使用字符串分割，打印出变量inp中存储的字符串前20个字符。

以这种方式读取文件时，所有的文本行和换行符被当做一个整体，作为一个大字符串存储在inp变量中。请记住，只有当计算机内存能够承载文件数据大小的情况下，才能用这种方式打开文件。

如果文件太大，内容无法承载，那就需要写一个for或while循环来分块读取。

7.5 搜索文件

搜索文件中数据的一种最常见方式是通读整个文件，只处理符合特定条件的文本行，其他一概忽略。我们将文件读取与字符串方法结合起来，构建简单的搜索机制。

举例来说，读取一个文件并把以“From:”开头的行打印出来。我们可以使用字符串startwith方法来选择符合前缀要求的行。

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:') :
        print line
```

程序运行结果如下：

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

输出结果正是我们想要的以“Form:”开头的行，但是为什么会有多余的空行出现呢？这是由于不可见的换行符所致。每一行都以换行符结束，因此print语句输出的变量line中的字符串带有一个换行符，print输出时本身还会增加一个换行符，所以最终就变成了空两行。

我们可以使用字符串分割来打印出不含最后一个字符的文本行，不过还有一个更简单的办法，使用rstrip方法截掉字符串后面的空白符，程序代码如下所示：

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:') :
        print line
```

程序运行结果如下：

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

随着文件处理程序变得越来越复杂，你可能会用到continue语句来编写搜索循环。搜索循环的基本思路是寻找“感兴趣的”行，跳过“不感兴趣的”行。当找到感兴趣的的文本行，执行相应的操作。

跳过不感兴趣的文本行的循环结构，代码如下所示：

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:') :
        continue
    # Process our 'interesting' line
    print line
```

程序运行结果是一样的。可以这样理解，不感兴趣的文本行就是那些不以“From:”开头的行，我们使用continue跳过这些行。对于感兴趣的文本行，也就是那些以“From：”开头的行，我们进行相应处理。

我们可以使用字符串方法find来模拟文本编辑器的搜索功能，找到任何一行中出现待查的字符串。由于find方法可以寻找一个字符串在另一个字符串中出现的次数，也可以返回字符串的位置或-1（表示字符串没有找到）。我们编写一个循环，找到包含“@uct.ac.za”字符串的文本行，也就是找到来自南非开普敦大学的邮件。

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1 :
        continue
    print line
```

程序运行结果如下：

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

7.6 让用户选择文件名

我们不希望在处理不同文件时每次都要修改python代码。一个更号的方法是每次运行程序时，让用户自己输入文件名。这样一来，用户就可以将这个程序用于不同文件，无需改动Python代码。

一个非常简单的方法可以满足以上需求，使用raw_input方法让用户输入文件名，代码如下所示：

```
fname = raw_input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print 'There were', count, 'subject lines in', fname
```

从用户那里获得文件名，把它赋予fname变量，然后打开对应的文件。现在，我们可以对不同的文件重复运行这个程序了。

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

进入下一小节之前，仔细查看这段程序，问问自己："可能会出现的错误是什么？"或者"友好的用户可能会做些什么，导致这个精巧的程序会报错，这样的话我们在用户眼中就不那么酷了"。

7.7 使用try、except与open

告诉过你不要偷看。这是最后的机会。

如果用户输入的不是一个文件名呢？

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'na na boo boo'
```

不要笑，不管是有意还是无意，用户做的任何操作都有可能破坏你的程序。事实上，软件开发团队中有一个重要的部分是质量保障（简称QA），可能由一个人或一个小组负责。他们的工作就是尽可能尝试破坏程序员开发的软件。

在用户购买软件或为程序员付薪水之前，QA团队的责任就是寻找程序的缺陷。可以这样说，QA团队是程序员的最佳搭档。

程序出错可以用try/except结构快速修复。假设open方法打开文件时会出错，那么在open方法打开失败时，增加一个恢复模式，代码如下：

```
fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print 'There were', count, 'subject lines in', fname
```

exit函数会终止程序，这个程序永不返回值。当用户（或QA团队）输入不正确的文件名时，我们可以"捕获"它们并快速进行修复：

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

对open方法调用的保护是Python程序中try/except的典型用法。以Python方式做事，我们使用"Pythonic"这个术语（Python风格）。上面的例子可称为文件打开的Python风格。当更加熟悉Python后，你会经常和其他程序员讨论，两个效果相同的方案哪一个更具有Python风格。愈发Python风格化的目的是将编程变得工程性与艺术性兼备。我们不仅要让程序能够正常工作，还要让解决方案更加优雅。

7.8 写入文件

为了能够写入文件，需要在打开文件时使用"w"作为第二个参数。

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

如果文件已经存在，以写入模式打开文件，这样会删除旧数据，因此请谨慎使用。如果文件不存在，那么会创建一个新的文件。

文件句柄对象的write方法把数据写入文件。

```
>>> line1 = 'This here's the wattle,\n'
>>> fout.write(line1)
```

在结束一行时，确保已明确插入了换行符。print语句会自动加上一个换行符，而write方法不会这样做。

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
```

当文件写入完成，记得关闭文件，确保写入物理磁盘，这样断电后数据才不会丢失。

```
>>> fout.close()
```

以读方式打开文件也要记得关闭文件。只顾打开新文件就显得有点粗心大意了。Python会在程序结束时，确认所有打开的文件被关闭了。当写入文件时，我们要对文件关闭进行明确声明，确保万无一失。

7.9 调试

读写文件时，你可能会遇到空格带来的问题。因为空格符、制表符和换行符是隐藏的，所以这些错误很难调试。

```
>>> s = '1 2\t 3\n 4'
>>> print s
1 2   3
 4
```

内置函数repr可以解决这类问题。它将任一对象作为参数，返回该对象的一个字符串表示。字符串中用反斜杠序列代表空白字符。

```
>>> print repr(s)
'1 2\t 3\n 4'
```

这对调试很有帮助。

另一个可能遇到的问题是，不同的操作系统使用不同的字符来表示一行的结束。一些操作系统使用换行符\n，一些操作系统使用返回字符\r，还有一些操作系统两者都使用。如果在不同的操作系统之间转移文件，这些差异可能会导致错误发生。

绝大多数操作系统都提供格式转换的应用。详细信息和更多疑问请访问 http://wikipedia.org/wiki/Newline。当然，你也可以自己写一个。

7.10 术语

捕获：使用try/except语句防止程序意外中止。

换行符：在文件和字符串中表示一行结尾的特殊字符。

Python风格：让Python更简洁、明确、高效工作的编程技术。使用try和except来恢复丢失的文件，这是Python风格的一个典型举例。

质量保证：负责软件产品整体质量保证的一个人或一个团队。QA任务包括产品测试与识别错误，一般在软件发布之前进行。

文本文件：保存在永久存储介质（如硬盘）的字符序列。

7.11 练习

习题**7.1** 编写一个程序，读取一个文件，以大写方式逐行打印出文件内容。程序运行结果如下所示：

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
  BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
  SAT, 05 JAN 2008 09:14:16 -0500
```

你可以从http://www.py4inf.com/code/mbox-short.txt下载文本文件。

习题**7.2** 编写一个程序，让用户输入文件名，然后读取文件，按行的形式j进行查看。

X-DSPAM-Confidence: 0.8475

遇到以“X-DSPAM-Confidence：”开头的行，提取该行中的浮点数。统计行数，计算这些行的垃圾邮件信度值。文件读取结束后，打印垃圾邮件平均信度。

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

用mbox.txt和mbox-short.txt这两个文件测试你的代码。

习题**7.3** 有时候，程序员感到无聊或是想找点乐子，他们会在程序里加入彩蛋（无害的代码）（http://en.wikipedia.org/wiki/Easter_egg_(media)）。修改上面的程序，当用户输入 “na na boo boo”时，打印一些有趣的消息。不管文件是否存在，程序都能正常运行。下面是程序运行样本：

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

不鼓励你在程序里加入彩蛋，这里只用作练习。

第8章 列表

8.1 列表即序列

与字符串类似，列表是由若干值组成的序列。字符串中的值是字符；列表中的值可以是任何类型。列表中的值称为元素或数据项。

列表的创建方法有好几种，其中最简单的是用方括号[]将元素括起来：

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

第一个例子是由四个整数组成的列表，第二个例子是由三个字符串组成的列表。列表的元素不要求是同一类型的。下面的列表包含一个字符串，一个浮点数，一个整数，以及另一个列表：

```
['spam', 2.0, 5, [10, 20]]
```

一个列表作为另一个列表的元素称为列表嵌套。

不含任何元素的列表称为空列表，使用空的方括号（[]）创建一个空列表。

正如你可能期望的，可以把列表值赋给变量：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2 列表是可变的

列表元素的访问与字符串中字符的访问语法是一样的，使用方括号操作符。方括号内的表达式指定索引位置。请注意，索引从0开始：

```
>>> print cheeses[0]
Cheddar
```

与字符串不同，列表是可变的。你可以改变列表中元素的顺序，或者对列表中的元素重新赋值。当括号运算符出现在赋值语句的左边时，就可以给列表中指定的元素赋值。

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

numbers列表的第二个元素之前是123，现在是5。

可以这样理解，列表是索引与元素之间的一种关系。这种关系称为映射，每一个索引对应一个元素。

列表与字符串的索引用法相同：

- 任何整数表达式都可作为索引。
- 试图读写一个不存在的元素时，你会得到IndexError索引错误提示。
- 如果索引值为负，表示从列表的尾部算起。

in运算符也适用于列表：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

8.3 列表的遍历

遍历列表元素最常用的方法是使用for循环。遍历语法与字符串遍历相同：

```
for cheese in cheeses:
    print cheese
```

如果只需遍历列表的元素，这个方法就足够了。但如果想写入或更新元素，这时就需要索引。一个常见方法是range函数和len函数的结合使用：

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

这个循环可以遍历并更新列表的每个元素。len函数返回列表中的元素个数。range函数返回一个索引列表（取值从0到n-1），其中n是列表的长度。对i进行循环，得到下一个元素的索引。函数体中的赋值语句使用i读取元素的旧值，然后给它赋予新值。

对于空列表来说，for循环不会执行函数体。

```
for x in empty:
    print 'This never happens.'
```

尽管一个列表可以包含另一个列表，但被包含的列表只能被看作一个元素。以下列表的长度为4：

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

8.4 列表的操作

“+”运算符连接多个列表：

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

类似地，“*”运算符对列表进行给定次数的重复：

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

第一个例子将列表[0]重复了四次，第二个例子将列表[1,2,3]重复了三次。

8.5 列表的分割

切片操作也适用于列表：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

如果省略第一个索引，切片将从列表头部开始；如果省略第二个索引，切片将处理到列表的末尾。如果两个索引参数都被省略，将会得到整个列表。

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

由于列表是可变的，在进行列表折叠、拉长和截断等操作之前，最好保留一份列表的副本。

赋值语句左边的切片操作可以更新多个元素：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

8.6 列表的操作方法

Python提供了一些操作列表方法。例如，使用append在列表尾部添加一个新元素。

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

extend可以将列表作为参数，并把一个列表的所有元素添加到另一个列表的尾部。

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

这个示例中t2没有发生改变。

sort可以将列表元素从低到高排序。

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

大多数列表方法是没有返回值的，它们会修改列表，但返回为None。如果你不小心写了下面这样的语句 `t = t.sort()` ，得到的t不是你所预期的排序。

8.7 删除元素

删除列表元素的方法有好几种。如果知道元素的索引位置，使用pop方法进行删除：

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
```

上面的程序使用pop修改了列表，打印输出了被删除的元素。如果没有指定索引位置，pop会删除并返回最后一个元素。

如果无需返回删除过的元素，可以直接使用del操作符。

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

如果只知道要删除的元素，但不知道它的索引位置，可以使用remove方法：

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

remove方法返回值为None。

当需要删除多个元素，可以根据切片索引，使用del来实现：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

一般认为，切片会删除两个索引位置之间的所有元素，实际上切片不会删除第二个索引参数所对应的元素。

8.8 列表与函数

列表有许多内嵌函数可用来遍历，无需另写循环代码：

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25
```

当列表元素为数字时，sum()函数才会起作用。其他函数如max()、len()等对字符串列表和其他可进行比较的数据类型才会起作用。

我们重写先前计算数字列表平均值的程序。用户输入一个列表，程序输出列表的平均值。

首先，不使用列表计算平均值的程序如下：

```
total = 0
count = 0
while ( True ) :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print 'Average:', average
```

在这个程序中，不断提示用户输入数据时，我们使用count和sum两个变量来记录数值个数以及求和。

我们只需记录下用户输入的数据，在循环结束后，利用内嵌的函数计算总和和个数。

```
numlist = list()
while ( True ) :
    inp = raw_input('Enter a number: ')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)

average = sum(numlist) / len(numlist)
print 'Average:', average
```

在循环开始之前，首先建立一个空列表，每得到一个数据，就把它添加到列表的尾部。在程序最后，只需将列表中数据的总和除以列表的长度，就可以求出平均值。

8.9 列表与字符串

字符串是字符的序列，而列表是一系列值的序列。字符列表与字符串是不同的。我们可利用list方法，把字符串转换成字符列表：

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

由于list是一个内置函数，应避免使用list作为变量名。笔者也会避免使用l作为变量名，因为容易和数字1搞混，这也就是为什么选择t作为变量名的原因。

list函数将一个字符串转化成一些单独的字母。如果想把一个字符串分成单独的单词，使用split函数：

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

一旦使用split函数将字符串分解成由单词组成的序列，你就可以利用索引操作符（方括号）来访问列表中的特定单词了。

split函数有一个可选参数，称为分隔符（delimiter）。它可以指定特定字符作为单词之间的界限。以下示例将连字符“-”作为分隔符：

join函数与split函数的作用相反。它使用字符串列表，把列表元素连接起来。join是字符串方法，所以必须指定分隔符，将列表作为参数。

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

在这个例子中，分隔符是一个空格。所以join函数会在两个单词之间加一个空格。如要连接字符串无需间隔，可以使用空字符串（''）作为分隔符。

8.10 行间解析

```
From stephen.marquard@uct.ac.za  Sat Jan  5 09:14:16 2008
```

split方法处理此类问题非常有效。先编写一个小程序，找到以“from”开头的句子，然后把这些句子分解，最后输出句子中的第三个单词：

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if not line.startswith('From ') : continue
    words = line.split()
    print words[2]
```

我们也可以通过在If语句的同一行放置continue，作为if语句的简化形式。从作用上讲，if函数的这种简化形式与continue在下一行的缩进形式是一样的。

程序运行结果如下：

```
Sat
Fri
Fri
Fri
    ...
```

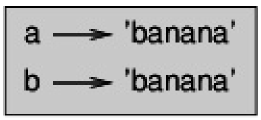
后续章节会介绍更复杂的行间文本提取技术，以及如何分解文本行，从中找到我们所需要的准确信息。

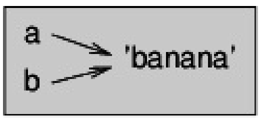
8.11 对象与值

执行以下赋值语句：

```
a = 'banana'
b = 'banana'
```

a和b都指向一个字符串，但它们所指向的是不是同一个字符串呢？存在以下两种情况：





一种情况是，a和b指向具有相同值的两个不同对象。另一种情况是，它们指向同一个对象。

为检验到底属于哪一种情况，使用is运算符;

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

在这个例子中，Python只创建了一个对象，a和b都指向它。

如果创建两个列表就会得到两个对象：

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

这种情况下，因为它们拥有相同的元素，我们可以说这两个列表是等价的，但不能说它们是同一个。它们不是相同的对象。当然，如果两个对象是同一个，那它们肯定是等价的，反之不成立。

截止目前，我们还在交替使用“对象”和“值”。更严谨的提法是，对象拥有值。当执行语句 `a = [1,2,3]` ，a会指向一个列表对象。这个列表对象的值是一个特定元素序列。此时，如果有另外一个列表与它的元素相同，那只能说它们拥有相同的值。

8.12 别名引用

如果a指向一个对象，当执行 `b=a` 后，两个变量都将指向同一个对象：

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

变量与对象之间的关系称为引用。在这个例子中，同一个对象有两个引用。

拥有多个引用的对象就会有多个名称，这种现象称作对象被赋予了别名。

如果别名化的对象是可变的，那么一个别名的变化，将会影响到其他别名的引用。

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

尽管这一行为是有用的，但也容易造成错误。一般而言，可变的对象最好不要使用别名。

像字符串这类不可变对象来说，别名并不会造成很大问题。如下所示：

```
a = 'banana'
b = 'banana'
```

a和b是否指向同一个字符串，这几乎没差别。

8.13 列表参数

当把一个列表传递给一个函数时，函数就会得到该列表的一个引用。如果这个函数改变了一个列表参数，调用者会知道这个变化。例如，delete_head删除了列表的第一个元素：

```
def delete_head(t):
    del t[0]
```

下面是它的用法：

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

上述参数t与变量letters是同一个对象的别名。

区分修改列表与新建列表的操作，这是非常重要的。例如，下面的示例中，append方法修改了列表，而“+”运算符新建了一个列表：

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None

>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
>>> t2 is t3
False
```

在编写列表的修改函数时，这种区别显得非常重要。例如，下面的函数就并没有达到删除列表头元素的目的：

```
def bad_delete_head(t):
    t = t[1:]              # WRONG!
```

切片操作会新建一个列表，赋值语句将t作为列表的引用。但这对于之前作为参数的列表来说没有任何影响。

另一种可行的办法是编写一个函数，创建并且返回一个新列表。例如，tail函数会返回一个新的列表，其包含除第一个元素之外的其他所有元素。

```
def tail(t):
    return t[1:]
```

这个函数并不会改变原始列表。以下是它的用法：

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

习题**8.1** 编写chop函数，移除列表的头元素和尾元素，并返回None值。

然后，编写middle函数，移除列表的头元素和尾元素，返回一个新列表。

8.14 调试

列表及其他可变对象的不谨慎使用，可能会造成长时间的调试。下面是一些常见陷阱以及如何避免的方法：

1） 大多数列表的方法会修改参数并返回None值，这与字符串的方法是相同。字符串的方法通常会返回一个新字符串，并不会改变原始的字符串。

如果你习惯如下的字符串代码的编写方式：

```
word = word.strip()
```

那你有可能写出如下的列表代码：

```
t = t.sort()           # WRONG!
```

因为sort函数返回None值，所以接下来赋予变量t会执行失败。

使用列表的方法和操作符之前，你应该仔细阅读文档，然后在Python的交互模式中测试。列表与其他序列（如字符串）共同拥有的方法和操作文档位于 http://docs.python.org/lib/typesseq.html。可变序列独有的方法与操作文档位于 http://docs.python.org/lib/typesseq-mutable.html。

2） 坚持并养成一种编写习惯

列表的部分问题是由于多种方式都可以达到相同目的造成的。比如，删除列表元素的方法有pop、remove、del甚至分片等。

若添加一个元素，你也可以使用append方法或“+”运算符。请不要忘记，下面的写法是正确的：

```
t.append(x)
t = t + [x]
```

而下面的写法是错误的：

```
t.append([x])          # WRONG!
t = t.append(x)        # WRONG!
t + [x]                # WRONG!
t = t + x              # WRONG!
```

请在Python交互模式中逐一练习，确保自己搞清楚了。还有，上面例子中只有最后一个会造成运行错误，其他三个可以运行，只不过没有按照我们的预期运行。

3）保留列表副本，避免直接引用

如果使用像sort这样会改变参数的方法，还要保留原始列表，复制一份列表副本。

```
orig = t[:]
t.sort()
```

在这个例子中，你也可以使用列表内置的sorted函数。这个函数会返回一个排序后的新列表，而不会改变原始列表。在这种情况下，应避免将sorted用于变量命名。

4） 列表、切片与文件

当读取和解析文件时，可能有一些输入会破坏我们的程序。编写一个程序来读取文件，进行“大海捞针”式找寻，这时打开守护模式是很有必要的。

让我们重温之前的行间读取程序，找出以From开头的行中包含的星期时间：

```
From stephen.marquard@uct.ac.za  Sat Jan  5 09:14:16 2008
```

由于我们将一行分解成了一个个单词，可以使用startwith函数，只需查看每一行的第一个单词，就可以决定它是否符合要求。另外，还可以使用continue来跳过不是以From开头的文本行，代码如下所示：

```
fhand = open('mbox-short.txt')
for line in fhand:
    words = line.split()
    if words[0] != 'From' : continue
    print words[2]
```

这看起来更简单了，甚至都不需要运行rstrip来删除文件末尾的换行符。但这样就真得更好了吗？

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

这个程序会部分运行成功，我们可以顺利地把第一个符合要求的行的第一个单词输出，但随后程序会中止，输出一个追踪错误。哪里出现问题了？到底是什么样的混乱数据，使得看似优雅灵活的Python程序出错？

你也许会较真，陷入深深的思考，或寻求他人帮助。最快捷和有效的方法是，添加一个print语句。添加语句的最佳位置在程序出错那一行之前，这样有可能打印出导致出错的数据。

虽然这个方法会输出大量的行，但至少可以快速找到问题解决的一些线索。因此，我们在第五句前面添加了一条打印变量words的语句。我们甚至还加了一个“Debug:”前缀，这样做是为了与其他正常的输出区分开来。

```
for line in fhand:
    words = line.split()
    print 'Debug:', words
    if words[0] != 'From' : continue
    print words[2]
```

运行这个程序，屏幕会出现大量滚动式输出。在输出的末尾，我们只需要查看调试输出和追踪错误信息，便可知道在追踪模块之前发生了什么。

```
Debug: ['X-DSPAM-Confidence:', '0.8475']
Debug: ['X-DSPAM-Probability:', '0.0000']
Debug: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if words[0] != 'From' : continue
IndexError: list index out of range
```

每一个调试行都会输出单词列表，这些单词是对文本行进行分解后得到的。当程序出错，单词列表为空[]。这时，在文本编辑器中打开文件，内容显示如下：

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

程序遇到了一个空行，错误原来在这里。很显然，空行里是没有单词的。我们在写代码时怎么就没有考虑到这个问题呢？当程序试着对空行的第一个单词（word[0]）进行“From”匹配时，由于没有找到单词，所以就出现“索引范围超出”的错误。

在第五行添加守护代码，避免对不存在第一个单词的空行进行比对，这个位置是最好的。还有许多方法来守护这个代码，我们会选择在查看第一个词之前，检查单词个数：

```
fhand = open('mbox-short.txt')
count = 0
for line in fhand:
    words = line.split()
    # print 'Debug:', words
    if len(words) == 0 : continue
    if words[0] != 'From' : continue
    print words[2]
```

首先，我们注释掉用于调试的print语句，不要直接删除它。这样做可以让修改失败时不必再重新调试。然后，我们添加一个守护语句，检查是否存在空单词，如果有的话，使用continue语句跳到文件的下一行。

这里使用了两个continue语句，过滤出我们感兴趣并希望继续处理的文本行集合。没有单词的行不是我们想要的，所以跳到下一行。不是以From开头的行也不符合要求，所以跳过。

修改后的程序能够成功运行，也许它就是正确的了。守护语句确实有效地避免了空行问题，但那也许还远远不够，当我们编写程序的时候，编程时我们必须经常思考“那里可能出问题了”。

习题**8.2** 找出以上程序中哪一行仍然没有得到有效守护。你也可以试着创建一个导致程序运行失败的文本文件，然后修改程序，以确保每一行都得到了相应的守护，确保该程序能够正确处理你创建的文本文件。

习题**8.3** 重写上例中的守护语句。使用复合逻辑表达式，在if语句中使用and逻辑运算符。

8.15 术语

别名：两个或多个变量指向同一个对象的情形。

分隔符：用来确定字符串如何被分割的字符或子字符串。

元素： 列表（或其他序列）中的一个值，也称为数据项。

等价：拥有相同的值。

同一性：同一个对象，可以推断出等价。

列表：一系列值的序列。

列表遍历：按顺序依次访问列表中的元素。

嵌套列表：一个列表作为另一个列表的一个元素存在。

引用：变量与其取值之间的关联。

8.16 练习

习题**8.4** 从http://www.py4inf.com/code/romeo.txt下载一个文本文件。

编写一个程序，打开romeo.txt文件，按行读取。对每一行使用split函数，将其分解成一系列的单词列表。对于每一个单词，检查它是否已经存在于列表之中，若单词还未出现在列表中，把它添加进来。

程序运行结束，按字母顺序输出最终的单词清单。

```
Enter file: romeo.txt
['Arise', 'But', 'It', 'Juliet', 'Who', 'already',
'and', 'breaks', 'east', 'envious', 'fair', 'grief',
'is', 'kill', 'light', 'moon', 'pale', 'sick', 'soft',
'sun', 'the', 'through', 'what', 'window',
'with', 'yonder']
```

习题**8.5** 编写一个程序，读取邮箱数据，当遇到一个以From开头的文本行，使用split函数将该行子分解成单词。我们需要抽取From开头的行中第二个单词，即发信人。

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

解析以From开头的行，打印出每行中第二个单词，还可以统计发信人数，在结尾输出总数。

下面是程序运行结果的部分输出：

```
python fromcount.py
Enter a file name: mbox-short.txt
stephen.marquard@uct.ac.za
louis@media.berkeley.edu
zqian@umich.edu

[...some output removed...]

ray@media.berkeley.edu
cwen@iupui.edu
cwen@iupui.edu
cwen@iupui.edu
There were 27 lines in the file with From as the first word
```

习题**8.6** 改写前面那个提示用户输入数据，并在用户输入“done”后，输出最大值与最小值的程序。编写一个程序，使用列表储存用户输入的数据，在循环结束后，利用max()和min()函数分别计算出最大值和最小值。

```
Enter a number: 6
Enter a number: 2
Enter a number: 9
Enter a number: 3
Enter a number: 5
Enter a number: done
Maximum: 9.0
Minimum: 2.0
```

第9章 字典

字典很像列表，但更通用。列表中的索引位置必须为整数，而字典中索引几乎可以是任意类型。

字典可看作是索引（这里称为键）的集合与值的集合之间存在的一种映射。每个键对应一个值，键与值之间的关系称之为键值对，有时也称为数据项。

举例来说，我们创建一个“英语-西班牙语”字典，其中键与值都是字符串。

dict函数可以创建一个空字典。请注意，由于dict是一个内置函数名称，不能把它用作变量名。

```
>>> eng2sp = dict()
>>> print eng2sp
{}
```

大括号{}表示一个空字典。你可以使用方括号向字典里添加数据项。

```
>>> eng2sp['one'] = 'uno'
```

这行语句创建了一个从键“one”到值“uno”映射的字典项。如果再次打印这个字典，我们会看到一个键值对，键与值之间用冒号隔开。

```
>>> print eng2sp
{'one': 'uno'}
```

字典的输入形式与输出形式是一样的。举例来说，你可以创建一个包含三个数据项的字典：

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

但是，如果你直接打印eng2sp，其结果会出人意料：

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

输出的键值对顺序发生了改变。事实上，如果你在计算机上输入这个字典，得到的输出结果跟上面的顺序可能也不一样。一般而言，字典的顺序往往是不可预测的。

即便如此，这不会造成什么问题。因为字典项并不是用整数来索引的，而是采用键来查找对应的值。

```
>>> print eng2sp['two']
'dos'
```

键“two”对应的值是“dos”，这与字典项的顺序无关。

如果查找的键不在字典里，你会得到一个异常提示：

```
>>> print eng2sp['four']
KeyError: 'four'
```

len函数也适用于字典，它会返回字典中键值对的个数。

```
>>> len(eng2sp)
3
```

in操作符也适用于字典，它会告知你查找的东西是否作为键存在于字典中，但不能对是否是值作出很好的判断。

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

为了判断要找的东西是否作为值存在于字典中，使用values方法，它会返回字典中所有值的一个列表，然后使用in操作符作出判断。

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

列表与字典的in操作符算法上是有差别的。在列表中，Python采用线性搜索策略，搜索时间与列表长度成正比。在字典中，Python采用了一种很有效的哈希表算法。这种算法不管字典里面有多少数据项，时间花费上几乎没有什么差别。此处不详述哈希表的原理，更多相关知识请访问http://wikipedia.org/wiki/Hash_table。

习题**9.1** 编写一个程序，读取words.txt文件中的单词，将它们作为键存储在字典中。此时不需要关心键对应的值。然后，使用in操作符快速判断某一字符串是否存在于字典中。

9.1 字典作为计数器

假设给定一个字符串，你想要知道其中每个字符出现的次数，以下几种方式可供选择：

1. 创建26个变量，分别记录字母表中每个字母出现的次数。然后，遍历字符串，每遇到一个字母，相应的计数变量加1，或许可以采用一个链式条件来实现。
2. 创建一个包含26个元素的列表。然后，使用Python内置的ord函数，将每一个字符转换成数字，把这个数字作为索引存入列表，再增加相应的计数变量。
3. 创建一个以字符为键，其出现次数作为值的字典。首次遇到一个字符，将其作为一个数据项添加到字典里。之后，遇到字典里已存在的数据项，就对其值加1。

以上方法都实现了相同的计算结果，但每种方法的计算策略各不相同。

执行计算的一种方式称为实现（implementation）。有些实现方式优于其他实现方式。举例来说，字典实现的一个优点在于，我们不必事先知道字符串中出现了哪些字母，只需为确实会出现的字母分配空间就好了。

下面是一段示例代码：

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print d
```

我们可以有效地计算出直方图。直方图是关于计数器（频次）的统计术语。

用for循环遍历字符串。每次循环时，如果字符c不在字典中，我们就创建一个字典项，以c为键，初始值1（表示这个字母至此出现了一次）。如果c已经存在于字典中，只需将对应值d[c]加1即可。

程序运行结果如下：

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

直方图表示字母“a”和“b”分别出现了一次，“c”出现了两次，后面的键值对含义类似，不再详述。

字典有一个get方法，该方法拥有一个键和一个默认值。如果键出现在字典中，那么它会返回此键对应的值。如果键不在字典中，返回事先给定的默认值。示例程序如下：

```
>>> counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
>>> print counts.get('jan', 0)
100
>>> print counts.get('tim', 0)
0
```

我们使用get方法可以把直方图循环写得更简洁。因为get方法自动处理了键不在字典中的情况。这样的话，代码的4条语句可缩减至1条，还可以去掉if语句。

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c,0) + 1
print d
```

使用get方法简化计数循环的做法已经成为Python中的一种普遍做法，本书后续章节会多次用到。因此，你仔细体会一下，if条件和in操作的循环与get方法的循环之间的区别。两者可以实现同样的目的，但是后者更加简洁。

9.2 字典与文件

字典的常见用法之一是对书面文字的文本文件进行词频统计。首先，我们从《罗密欧与朱丽叶》中抽取一段简单的文本，参照 http://shakespeare.mit.edu/Tragedy/romeoandjuliet/romeo_juliet.2.2.html。

由于刚开始接触，我们使用不包含标点符号的简化后的短文本。随后会介绍包含标点符号的文本处理方法。

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

编写一个Python程序，读取上述文件的每一行，并将每一行分解为由单词组成的一个列表。然后，遍历每个单词，使用字典来统计每个单词的出现次数。

你可以看到，这段程序包含两个for循环，外部循环用来读取文件中每一行，内部循环迭代出该行的每一个单词。这就是所谓的嵌套循环，其中一个为外部循环，另一个为内部循环。

由于外部循环每迭代一次，内部循环都会执行它全部的迭代。基于这一点，我们认为内部循环比外部循环迭代地“更快”。

两个嵌套循环的组合确保了输入文件的每一行的每个单词都会被统计到。

```
fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts
```

运行程序，生成一个包含全部计数的原始文件，输出没有排序的哈希序列。romeo.txt文件可在http://www.py4inf.com/code/romeo.txt获得。

```
python count1.py
Enter the file name: romeo.txt
{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,
'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,
'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,
'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,
'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}
```

通过字典找到出现最多的单词及其次数，并不是那么方便。因此，我们需要增加一些Python代码，以便输出更有用的信息。

9.3 循环与字典

如果在for语句中把字典看做一个待循环的序列，它会遍历字典中的每一个键。下面的循环输出每一个键及其对应的值：

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    print key, counts[key]
```

程序运行结果如下：

```
jan 100
chuck 1
annie 42
```

再次强调，字典中的键是没有固定顺序的。

我们可以将这种模式应用到之前介绍的循环的各种常用方法。例如，如果想要找到字典中所有值大于10的记录，程序代码如下：

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
for key in counts:
    if counts[key] > 10 :
        print key, counts[key]
```

for循环是通过字典的键进行迭代。因此，我们必须使用索引操作符来检索相对应的值。程序运行的输出结果如下：

```
jan 100
annie 42
```

我们只看到了值大于10的记录。

如果按照字母顺序输出字典中的键，首先使用字典对象的keys方法，将所有的键放入一个列表。然后，对这个列表进行排序，对排序后的列表进行迭代。依照字母顺序，通过每个键输出对应的键值对：

```
counts = { 'chuck' : 1 , 'annie' : 42, 'jan': 100}
lst = counts.keys()
print lst
lst.sort()
for key in lst:
    print key, counts[key]
```

程序运行的输出结果如下：

```
['jan', 'chuck', 'annie']
annie 42
chuck 1
jan 100
```

首先，通过keys方法得到了包含键的未排序的列表。然后，通过for循环得到有序的键值对。

9.4 高级文本解析

上面示例程序用到的romeo.txt文件是我们手动删除所有标点符号之后得到的尽可能简化的文本。实际的文本会包含大量标点符号，如下所示：

```
But, soft! what light through yonder window breaks?
It is the east, and Juliet is the sun.
Arise, fair sun, and kill the envious moon,
Who is already sick and pale with grief,
```

Python的split函数可以识别空格，把词汇看作是由空格分隔开来的词单元，所以，“soft”和“soft!”会被视为不同的词汇，分别为它们创建一个字典项。

由于文本中还存在大写，“who”和“Who”也是不同的词，分别进行统计。

通过字符串的lower、punctuation与translate方法可以解决上述问题。其中translate是最精细的方法。以下是translate的说明文档：

string.translate(s, table[, deletechars])

首先，从*s*中删除*deletechars*参数（如果存在的话）指定的所有字符，然后使用*table*参数来翻译字符。*table*是一个*256*个字符长的字符串，用来翻译每个字符值，并按照序数进行索引。如果*table*是*None*值，只会执行字符删除步骤。

这里不指定table参数，用deletechars参数来删除所有标点符号。Python甚至可以告诉我们，标点符号包括哪些字符：

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

我们对程序做出如下修改：

```
import string                                                  # New Code

fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)    # New Code
    line = line.lower()                                # New Code
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

print counts
```

我们使用translate方法删除了所有的标点符号，并将每一行中的字母转换为小写。程序主体并未发生改变。请注意，Python2.5及早期版本中，translate方法不接受None值作为第一个参数，因此使用下面的代码来调用该方法：

```
print a.translate(string.maketrans(' ',' '), string.punctuation
```

懂得一些“Python的艺术”和“像Python一样思考”，不难发现Python为许多常见数据分析问题内置了解决方案。随着学习的深入，通过大量的示例代码和技术文档的阅读，你会知道去哪里寻找别人是否用Python已经解决了此类问题，从而减轻一些你的工作。

以程序运行的部分输出结果如下：

```
Enter the file name: romeo-full.txt
{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,
'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,
a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,
'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,
'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}
```

查看这些输出仍然很费事，让Python来帮助我们找到具体要找到的信息。要做到这一点，我们需要学习Python的元组。在学习元组时会继续使用这个例子。

9.5 调试

当需要处理更大的数据集时，手工输出与检查数据显得不那么现实了。以下是调试大数据集的一些建议：

减少输入：尽可能地减少数据量的大小。例如，程序读取一个文本文件，仅选择前十行，或者选择最小的示例。你可以编辑文件本身，或者修改程序，让它只读取文件的前n行。后一种方式更好一些。

当出现错误时，通过不断地减小n值来确定错误的位置。当找到错误并纠正后，再不断增大n值。

检查摘要与类型：不要对整个数据集进行输出和检查，可以考虑先输出数据的摘要。例如，字典的数据项个数，或数字列表的总数。

运行错误的一个常见原因是数据类型不正确。调试此类错误，通常输出这个值的类型就可以解决了。

编写自检程序：有时通过编写代码来自动检查错误。例如，需要计算数字列表的平均数，你可以检查这个结果是不是处于列表的最大值与最小值之间。这种检查称为“逻辑检查”，它会检测出哪些结果是完全不符合逻辑的。

另一种检查是比较两种不同计算的结果，检查它们是否一致。这种检查称为“一致性检查”。

工整化输出结果：对调试的输出结果进行格式化，这有助于发现错误。

再次强调，在程序架构上花些心思能有效减少调试的时间花费。

9.6 术语

字典：一组键及其对应值的映射。

哈希表：Python字典的实现算法。

哈希函数：使用哈希表来计算字典中键的位置的函数。

直方图：一组计数器。

实现：执行计算的一种方法。

数据项：键值对的另一种说法。

键：字典中键值对的第一部分对象。

键值对：从键到值的映射关系表达。

查找：字典的一种操作，根据键找到对应的值。

嵌套循环：一个循环内部包括另一个或多个循环。外部循环每执行一次，内部循环会全部执行一遍。

值：字典中键值对的第二部分对象。这里所说的值比之前提到的“取值”更加专指。

9.7 练习

习题**9.2** 编写一个程序，按照接收的星期时间对邮件进行分类。首先，找出以From开头的文本行，然后，取出符合条件的每一行中的第三个单词，使用一个计数器统计一周中每一天邮件被接收的次数。在程序的末尾，输出字典的内容（不要求次序）。

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Sample Execution:
python dow.py
Enter a file name: mbox-short.txt
{'Fri': 20, 'Thu': 6, 'Sat': 1}
```

习题**9.3** 编写一个程序，读取邮件日志，使用字典来创建一个直方图，统计每个邮箱发出的邮件数量，然后输出字典。

```
Enter file name: mbox-short.txt
{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,
'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,
'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,
'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,
'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,
'ray@media.berkeley.edu': 1}
```

习题**9.4** 在上述程序中添加代码，从文件中找出谁的邮件最多。 读取完所有数据并且建立字典之后，使用最大循环（详见5.7.2）对字典进行遍历，找出谁的邮件最多，并输出他的邮件总数。

```
Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

习题**9.5** 编写一个程序，记录邮件发送者的域名，而不是邮件发送者的邮件地址。在程序的末尾，输出字典的内容。

```
python schoolcount.py
Enter a file name: mbox-short.txt
{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,
'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}
```

第10章 元组

10.1 元组是不可变的

元组[1]是由若干值组成的一个序列，与列表非常相似。元组中存储的值可以是任何数据类型并拥有整数型索引。元组的一个重要特征是不可变的。元组可以进行比较和使用哈希算法，我们可以对其进行排序，在Python字典中，使用元组作为键值。

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

元组用圆括号括起来，虽然这不是必需的，但可以帮助我们在Python代码中快速识别出哪些是元组。

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

创建单个元素的元组，必须在末尾加一个逗号。

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

如果没有逗号，Python会将(' a ')作为一个字符串处理：

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

构造元组的另一种方法是，使用内置函数tuple。如果不带参数，tuple函数会创建一个空元组：

```
>>> t = tuple()
>>> print t
()
```

如果参数是一个序列（字符串、列表或元组），tuple函数的调用结果是产生一个包含序列元素的元组：

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

由于tuple是构造器的名称，应避免将其用作变量名。

大部分的列表操作符也适用于元组。方括号索引一个元素：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

切片操作符选择一组元素。

```
>>> print t[1:3]
('b', 'c')
```

但是，如果尝试修改元组中的元素，会得到一个错误：

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

不能修改元组的元素，但可以用另一个元组替换当前元组：

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

10.2 元组的比较

比较运算符适用于元组和其它序列，Python从每个序列的第一个元素开始比较。如果它们相等，继续比较下一个元素，以此类推，直到找到不同的元素。找到不同元素之后，随后的元素就不再考虑了（即便它们真得很大）。

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

元组中sort函数正是这个工作原理。它首先对第一个元素排序，如果第一个元素相同，则按第二个元素排序，以此类推。

这一特性使其拥有一种DSU模式：

修饰（**Decorate**）：修饰：在序列的元素之前放置一个或多个排序键，

排序（**Sort**）：使用Python内置函数sort进行排序，

去修饰（**Undecorate**）：提取出序列中已排序的元素。

举例来说，有一组单词，对它们进行由长到短的排序:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
   t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print res
```

第一个循环创建了一个元组，每个元组包括单词及长度，单词长度在前，单词在后。

sort函数进行两两比较，首先比较单词长度，如果长度相等，则比较元组中的第二个元素。关键字参数reverse=True指定sort函数按照倒序排列。

第二个循环遍历了元组，创建了一个按照长度降序排列的单词列表。这五个单词按照反向字母顺序排序，所以“what”出现在“soft”之前。

程序输出结果如下:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

当然，这一行文字被转换为Python列表，并按照单词长度降序排列之后，它就失去了原有的诗歌意味[2]。

10.3 元组的赋值

Python语言的一个独特句法特征是，元组可以出现在赋值语句的左侧。当左侧是一个序列时，一次可以为多个变量赋值。

在本例中，一个列表（序列）包含两个元素。使用一行语句，将第一个元素和第二个元素分别赋予变量x和变量y。

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

这不是魔法，Python会大致翻译元组的赋值语法，如下所示：[3]

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

从文体上看，在赋值语句左侧使用元组，我们忽略了括号，以下是有效的等价语法：

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

元组赋值有一个特别巧妙的用途，可以在一条语句中交换两个变量的值：

```
>>> a, b = b, a
```

这条语句两侧都是元组，左侧是变量元组，右侧是表达式元组。右侧的每个值分别赋予左侧的每个变量。右侧的所有表达式在赋值之前进行检查。

左侧的变量个数与右侧的值个数必须相同：

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

更普遍的情况是，右侧可以是任何类型的序列，如字符串、列表或元组。例如，将邮件地址拆分成用户名与域名的程序代码如下：

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

split的返回值是包含两个元素的列表。第一个元素是uname，第二个元素是domain。

```
>>> print uname
monty
>>> print domain
python.org
```

10.4 字典与元组

元组拥有items方法，该方法返回元组列表，每个元组是一个键值对[4]。

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> print t
[('a', 10), ('c', 22), ('b', 1)]
```

如果是字典的话，其中的数据项是没有特定顺序的。

由于元组列表本身是一个列表，元组之间可以进行比较，以及对元组列表进行排序。将字典转化为元组列表，这样可以根据键对字典进行排序，并输出字典的内容。

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
>>> t
[('a', 10), ('c', 22), ('b', 1)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

新的列表根据键值以字母升序排列。

10.5 通过字典进行多个赋值

将items方法、元组赋值与for循环结合起来，你可以拥有一个良好的代码模式，使用单循环就可以遍历字典中的键与值。

```
for key, val in d.items():
    print val, key
```

这个循环中存在两个迭代变量。由于items返回一个元组列表，变量key和val通过字典的键值对进行迭代，继而得到赋值。

循环中的每次迭代都会使得key和value被赋予下一个字典键值对（仍然以哈希顺序）。

此循环的输出结果如下：

```
10 a
22 c
1 b
```

再次强调，这是哈希键顺序，也就是没有特定顺序。

将两种方法结合，按照每个键值对中的值来排序，输出字典的内容。

要做到这一点，首先创建一个元组列表，其中每个元组为(value, key)。通过items方法得到(key, value)元组列表。此时，我们想要根据value排序，而不是key。(value, key)元组列表一旦生成，排序就变得简单了，按照反向次序对列表排序，输出已排序的新列表。

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

创建元组列表时要非常谨慎，确保每个元组的第一个元素是值，这样就能对元组列表进行排序，获得所需的字典内容，该字典已按值进行排序。

10.6 高频词汇

回头看看前面介绍的单词统计程序，文本取自《罗密欧于朱丽叶》的第二幕第二场。现在，我们扩充这个程序，输出文本中出现次数最多的前十个单词，代码如下所示：

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(None, string.punctuation)
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
lst = list()
for key, val in counts.items():
    lst.append( (val, key) )

lst.sort(reverse=True)

for key, val in lst[:10] :
    print key, val
```

程序的第一部分读取文件，计算出文档中每个单词出现的次数，将单词及其出现次数放入字典中。这部分程序不做修改。之前输出变量count的值之后，程序就结束了，这里我们创建一个(val, key)元组列表，按照逆序对列表进行排序。

由于元组中值是第一个元素，所以它被用于比较。如果多个元组拥有相同的值，接下来检查元组的第二个元素（键）。因此，如果值相同，元组将按照键的字母顺序进行排序。

在程序末尾，我们写了一个for循环，实现多个赋值的迭代，通过列表切片操作（lst[:10]），迭代输出前十个高频词汇。

至此，程序的输出看上去符合我们想要的词频分析结果。

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

事实上，如此复杂的数据解析与分析只需19行易于理解的Python代码就解决了。这就是Python语言用于信息分析的明智选择依据之一。

10.7 在字典中使用键作为元组

元组可以使用哈希算法排序，但列表不可以。如果我们想在字典中创建一个组合键，那么必须使用元组作为键。

假设创建一个电话号码簿，包含姓名与电话号码的对应关系，那么就会用到组合键。若已经定义了变量last、first与number，字典赋值语句如下：

```
directory[last,first] = number
```

方括号里的表达式是一个元组。在for循环中使用元组赋值来遍历这个字典。

```
for last, first in directory:
    print first, last, directory[last,first]
```

这个循环遍历了directory中的键，这里的键是元组。它给每个元组的元素赋予变量last和first，然后打印出姓名与对应的电话号码。

10.8 序列：字符串、列表与元组 —— 哦，这么多！

此处的重点是元组列表，但本章几乎所有示例都可以用于列表的列表，元组的元组以及列表的元组。为避免列举过多，有时为方便统称为序列的序列。

在许多情况下，不同类型的序列（字符串、列表与元组）之间可以互换使用。既然这样，为什么要选择这一种序列而不用其他序列？另外，如何选择合适的序列呢？

从最明显的字符串说起，由于字符串元素只能是字符，所以它比其他序列的局限性更大。另外，字符串是不可改变的。如果需要在字符串中修改字符，而不是新建一个字符串，那么可能需要使用字符列表。

列表比元组更为常见，主要是因为列表是可变的。以下情况可能更适合元组：

某些情况下，例如return语句，它创建元组的语法比创建列表的要简单。另一些情况下，可能列表更合适。

如果使用序列作为字典的键，那么必须使用不可变类型，如元组或字符串。

如果将序列作为参数传递给函数，那么使用元组会减少由于别名带来意外情况的可能。

因为元组是不可变的，所以没有sort和reverse这类可以修改已有列表的方法。然而，Python提供了内建函数sorted与reversed，将任一序列作为参数输入，之后返回一个元素相同但次序不同的新序列。

10.9 调试

列表、字典与元组通常被称为数据结构。本章中我们见识到了复合数据结构，例如，元组列表，以元组为键、列表为值的字典。复合数据结构是有用的，但容易出现形状错误，即由于数据结构的类型、大小或组成出错所致，亦或编写代码时由于忘记数据类型导致出错。

举例来说，如果列表包含一个整数，给出一个不在列表里的其他整数，这就不起作用。

调试程序时，如果遇到困难，可以尝试以下四种方法：

阅读：检查代码，仔细阅读，验证代码是否按照你的意愿执行。

运行：通过修改代码进行实验，运行不同版本的代码。通常，程序在正确的地方显示了正确的东西，那么程序显而易见，但有时候得花些时间来构建程序的支架。

沉思：花一些时间去思考！错误类型究竟是什么？语法、执行时或语义？从错误消息或程序输出中能得到什么信息？什么样的错误类型会导致你正遇到的问题？在问题出现之前，最后一次你做了什么？

回退：某些时候最好的办法是回退，撤销最近的修改，退回到程序可以正常工作和你能够理解的状态。之后开始重建。

编程新手有时会陷在这些活动之中，忘记其他事情。每种活动都有自身的故障模式。

举例来说，如果程序存在输入错误，只要问题不是概念上的误解，阅读代码就可以解决。如果不明白程序做了什么，即使阅读100遍也看不到错误，因为错误就在你的脑袋里。

运行实验可以提供帮助，特别是一些小型简单的测试。然而，如果你没有经过思考或仔细阅读代码就运行实验，那么可能会陷入一种“随机游走编程”模式，也就是说通过随机修改，直到程序正确执行。毫无疑问，随机游走编程会花费很长时间。

你必须花时间去思考。调试就像一门实验科学。你应该至少有一个关于问题的假说。如果存在两个或更多可能时，试着考虑用测试去消除其中之一。

休息一下有助于思考。与人交谈也一样。如果你向其他人（甚至是自己）解释问题，有时在问完问题之前，你就找到答案了。

但是，如果错误太多或尝试修复的代码过于庞大和复杂，即使最好的调试技术也无济于事。有时最好的选择是以退为进，对程序进行简化，直到你能掌控和理解的程度。

编程新手往往不愿意回退，因为他们无法忍受删除代码（即使它是错的）。如果你对自己写的代码感觉还不错，在删除代码之前复制一份到另一个文件中。之后，你就能每次粘贴回来一小部分代码。

找到难度大的错误需要阅读、运行、沉思以及时而后退等步骤。如果你陷入其中一种，试试其他的。

10.10 术语

可比较的：相同类型的值之间进行比较，包括大于、小于或等于。可进行比较的类型可以放在列表中，并可以排序。

数据结构：相关值的集合，常见形式有列表、字典、元组等。

DSU: “修饰-排序-去修饰”的缩写，包括构建元组列表、排序与抽取部分结果的一种模式。

聚集：组装变长参数元组的操作。

可哈希的：拥有哈希函数的类型。不可变类型，如整数、浮点数和字符串，都可以使用哈希函数；可变类型，如列表与字典，不可以使用哈希函数。

散布：将序列视为参数列表的操作。

数据结构的形状：数据结构的类型、大小与组成的概要。

单例：包含一个元素的列表（或其他序列）。

元组：不可变的元组序列。

元组赋值：右侧是序列赋值，左侧是变量元组。右侧通过检验后，将其中的元素赋予左侧的变量。

10.11 练习

习题**10.1** 修改之前的程序：读取与解析出以“From”开头的行，取出符合条件的行中的邮箱。使用字典统计出每个人的邮件数。

当所有的数据读取完毕，从字典中创建一个元组(count, email)列表，然后对列表进行反向排序，打印出提交最多的用户。

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

习题**10.2** 统计出每条消息在一天中的小时分布。从以“From”开头的行中，找出时间字符串，根据冒号将其分解。对每个小时的次数进行累积，按行打印出统计数，并按照小时进行排序，程序运行结果如下所示：

```
Sample Execution:
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

习题**10.3** 编写一个程序，读取一个文件，按照频率降序打印出字母。这个程序将所有输入转化成小写，只统计字母a-z，不统计空格、数字、标点符号以及其他a-z之外的字符。尝试使用不同语种的文本片段，观察不同语言之间的词频差异。将结果与http://wikipedia.org/wiki/Letter_frequencies上的词频表进行比较。

> 1. 有趣的事实："元组"的命名取决于序列的长度，如一元组（single）、二元组（double）、三元组（triple,）、四元组（quadruple）、五元组（quintuple）、六元组（sextuple）七元组（septuple）等。↩

> 2. 译者注：此句出自莎士比亚的《罗密欧与朱丽叶》。原文为"But, soft! what light through yonder window breaks? "，本书作者略作修改，将through改为in。↩

> 3. Python不做字面上的语法翻译。如果你使用Python的字典进行语言翻译，无法得到想要的结果。↩

> 4. Python 3.0中的用法有些许差异。↩

第11章 正则表达式

至此，我们已经学会如何读取文件，寻找模式与提取感兴趣的文本行片段。文本行提取的方法，包括字符串方法（如split与find）、列表与字符串切片等。

文本搜索与抽取是常见任务。Python拥有非常强大的快速有效处理此类任务的库——正则表达式。正则表达式功能强大，比较复杂，其语法需要一些时间来熟悉，所以本书并没有过早提及。

正则表达式几乎就是一门关于字符串搜索与解析的小型编程语言。事实上，整本书都是围绕正则表达式主题展开。本章仅介绍正则表达式基础，有关正则表达式更多详细信息，请参阅以下网址：

http://en.wikipedia.org/wiki/Regular_expression

http://docs.python.org/library/re.html

正则表达式库在使用之前必须先导入到程序中。正则表达式库最简单的用法是search()函数。搜索函数的简单用法如下程序所示：

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line) :
        print line
```

打开文件，循环每一行，使用正则表达式的search()函数，打印出包含字符串"From:"的文本行。这个程序其实并没有发挥正则表达式的真正实力，line.find()函数可以更容易地实现相同的结果。

正则表达式的强大之处体现于，可以在搜索字符串中添加特定字符，以实现更精确的字符串文本行的匹配控制。通过在正则表达式中添加特定字符，编写很少代码就可以实现复杂的匹配与抽取。

例如，正则表达式的^符号匹配一行的开始。我们修改一下上面的程序，仅匹配“From:”开头的文本行。

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line) :
        print line
```

好了，这就做到仅匹配“From:”开头的文本行。这仍然是一个非常简单的例子，字符串库的startswith()函数同样可以实现。之所以这样讲解，目的是介绍正则表达式的理念，包含特定行动字符，给予文本匹配更多的控制。

11.1 正则表达式的字符匹配

许多特定字符可以帮助我们编写非常强大的正则表达式。最常用的特定字符是句点，它可以匹配所有字符。

在下面的例子中，正则表达式"F..m:"会匹配配“From:”、"Fxxm"、“F12m”或"FI@m"。正则表达式的句点可以匹配任意字符。

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line) :
        print line
```

“*”和“+”表示一个字符可以重复任意次数，在构造正则表达式时结合这种能力特别有用。这些特定字符用来代替单个字符，星号匹配零或多个字符，加号匹配一个或多个字符。

进一步减少代码，以下示例使用重复的通配符：

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line) :
        print line
```

搜索字符串“^From:.+@”会成功匹配以“From:”开头，之后一个或多个字符，以@结尾的文本行。结果匹配如下所示：

```
From: stephen.marquard @uct.ac.za
```

可以这样理解，“.+”通配符匹配了冒号与@之间的所有字符。

```
From:.+ @
```

有时候，加号与星号可能会“用力过猛”。例如下面的字符串匹配，".+"将其外推，直到最后一个@。

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu
```

通过添加其他字符，让星号和加号不要如此“贪婪”地匹配，这是可以做到的。详情请参阅“关闭贪婪行为”的文档。

11.2 使用正则表达式抽取数据

在Python中抽取字符串的数据，用到的是findall()函数。通过正则表达式的匹配，抽取所有符合的子字符串。以下示例从格式无关的任何文本行中抽取类似电子邮件地址的文本。

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

我们不想为每个文本行类型编写代码，每个文本行都分割和切片一次。以下程序使用findall()找到文本中的电子邮件地址，从每一行抽取一个或多个电子邮件地址。

```
import re
s = 'Hello from csev@umich.edu to cwen@iupui.edu about the meeting @2PM'
lst = re.findall('\S+@\S+', s)
print lst
```

findall()函数搜索第二个参数的字符串，返回一个包含形如电子邮件地址字符串的列表。我们使用两字符序列来匹配非空字符(\S)。

程序运行结果如下：

```
['csev@umich.edu', 'cwen@iupui.edu']
```

解释一下这个正则表达式，我们寻找至少含有一个非空字符的子字符串，之后是@，然后再是至少一个或多个非空字符。“\S+”匹配尽可能多个非空字符。这就是正则表达式中的贪婪匹配。

正则表达式会匹配两个电子邮件地址，但不会匹配“@2PM”，原因是@之前没有非空字符。在程序中使用这个正则表达式，读取文件的所有行，然后打印出所有类似电子邮件地址的结果，如下所示：

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0 :
        print x
```

读取每一行，抽取与正则表达式匹配的所有字符串。由于findall()返回的是列表，我们简单查看下返回的列表不为零，打印出来的每行至少包含一个电子邮件地址。

对mbox.txt运行程序，得到如下结果：

```
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['<postmaster@collab.sakaiproject.org>']
['<200801032122.m03LMFo4005148@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost]']
['source@collab.sakaiproject.org;']
```

一些电子地址的开头或结尾包含了不正确的字符，如“<”或“;”。这里声明一下，仅需要以字母或数字开头和结尾的字符串部分。

要做到这一点，我们使用正则表达式的另一个功能，使用方括号罗列多个可接受的匹配字符。在某种意义上，“\S”匹配的是非空字符的集合。现在，我们更清楚一些字符匹配的本质了。

下面是新的正则表达式：

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

这看起来有点复杂，你现在应该明白正则表达式为什么被称为一门专门的语言了。解释一下这个正则表达式，寻找以一种子字符串，以小写字母、大写字母或数字开头[a-zA-Z0-9]，之后是零个或多个非空字符“\S”，然后是@，再是零个或多个非空字符“\S”，最后是一个大写或小写字母。请注意，我们从加号到星号，再到零个或多个非空字符。[a-zA-Z0-9]本身就是一个非空字符。请记住，星号和加号直接作用于它左侧的单个字符。

如果在程序中使用这个正则表达式，数据会变得干净一些：

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
    if len(x) > 0 :
        print x
...
['wagnermr@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

注意到source@collab.sakaiproject.org这一行，正则表达式消除了字符串结尾（“>”）结尾的两个字母。原因是我们在正则表达式末尾追加了“[a-zA-Z]”，要求正则表达式解析器对找到的字符串必须以字母结尾。因此，当出现“sakaiproject.org>;”，它会止步于匹配找到的最后一个字母，这里g是最后一个符合要求的字符匹配。

还要注意的是，该程序的结果是一个Python列表，每个字符串是一个元素。

11.3 将搜索与抽取结合

如果我们想要找到以“X-”开头的文本行，如下所示：

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

我们不仅需要文本行中的浮点数，还需要统计符合以上语法的文本行数。

使用下面的正则表达式来挑选出符合要求的文本行：

```
^X-.*: [0-9.]+
```

解释一下，文本以“X-”开头，之后是零个或多个字符“.*”，然后是一个冒号和一个空格。空格之后是一个或多个字符，可以是一个数字(0-9)或一个句点“[0-9.]+”。需要注意的是，方括号中的句点实际匹配的是句点本身，也就是说，它在方括号内不是通配符。

这是一个非常紧凑的表达式，我们感兴趣的文本匹配如下所示：

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line) :
        print line
```

运行这个程序，经过过滤的数据仅保留如下内容：

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
```

现在，我们要解决抽取数值的问题，使用split方法。虽然使用split很简单，我们这里使用正则表达式的另一个功能，让搜索与解析同时进行。

括号是正则表达式的另一个特殊字符。在正则表达式中添加括号，括号的内容将在匹配时被忽略。但是，在 findall()函数中括号表示的是匹配括号内的整个表达式。在抽取与正则表达式匹配的子字符串部分，findall()函数适用。

这样，修改之后的程序代码如下：

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0 :
        print x
```

与search()函数不同，我们在正则表达式中添加括号来表示浮点数，指明我们只需要findall()函数找出匹配到的字符串的浮点数。

程序运行结果如下：

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
```

数字仍然是存在列表中，需要把字符串转换为浮点数。这里侧重展示正则表达式可以同时进行搜索与抽取的功能实现。

如果文件包含如下形式的文本行，使用这种方法的另一个例子如下：

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

如果想要抽取所有的修订号（每一行末尾的整数值），程序代码如下：

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9]+)', line)
    if len(x) > 0:
        print x
```

解释一下这个正则表达式，以“Details:”开头，之后是任意字符“.*”，然后是“rev=”，最后是零或多个数字。我们只需要文本行最后的整数值，所以用括号把[0-9]+括起来。

程序运行结果如下：

```
['39772']
['39771']
['39770']
['39769']
...
```

请记住，“[0-9]+”是“贪婪的”，在抽取这些数字之前，它试图匹配尽可能多的符合条件的字符串。这个贪婪行为是获得5位数字的原因所在。正则表达式库进行了前后扩展，直到它在开头或结尾遇到一个非数字才停止匹配。

现在，我们可以使用正则表达式重做之前的邮件消息中的时间提取。文本内容如下：

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

此处抽取每一行中当天的小时。之前的做法是调用split两次。第一次，文本行分解为单词，取出第五个单词，将其再次用冒号分解。最后，取出我们需要的前两个字符。

虽然这样做达到了目标，但在代码编写时缺乏一定灵活性，前提是文本需要经过良好的格式化。如果增加足够的错误检查或一大块的try/except代码，确保程序在遇到格式不正确的文本行时不会出错，代码会增长到10-15行，那就不太好阅读了。

使用下面的正则表达式可以更容易地做到这一点：

```
^From .* [0-9][0-9]:
```

解释一下这个正则表达式，以“From ”开头（注意空格），之后是任意多个字符“.*”，然后空一格，接着是2位数字“[0-9][0-9]”，最后是一个冒号。这样的定义符合之前想要寻找的内容。

为了只取出小时数，使用findall()方法，在两位数字上加括号，正则表达式如下：

```
^From .* ([0-9][0-9]):
```

程序代码如下：

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0 : print x
```

程序运行结果如下：

```
['09']
['18']
['16']
['15']
...
```

11.4 转义字符

由于在正则表达式中使用特殊字符来匹配一行的开头与结尾，或指定通配符，那么需要一种方法来保证这些特殊字符本身的指代性，例如匹配$与^符号本身。通过在字符前使用反斜杠作为前缀可以轻松解决这个问题。例如，使用以下正则表达式找出金额数。

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

由于$符号之前有一个反斜杠，它实际上匹配的是美元符号本身，不是匹配一行的结尾，正则表达式的其他部分匹配一个或多个数字和句点。请注意，方括号内，字符没有特殊性。因此，[0-9.]实际表示数字和句点。方括号之外，句点是一个通配符，匹配任意字符。在方括号之内，句点就代表它本身。

11.5 小结

虽然本章只触及了正则表达式的皮毛，但我们已经对正则表达式这门语言有所了解。包含特殊字符的搜索字符串能够按照意愿，构建正则表达式来定义匹配的字符和想要抽取的内容。以下是一些特殊字符和字符序列：

^ 匹配文本行的开头。

$ 匹配文本行的结尾。

. 匹配任一字符（一个通配符）。

\s 匹配一个空白字符。

\S 匹配一个非空字符（与\s相反）。

* 应用于前接字符，表示前接字符的零个或多个匹配。

*? 应用于前接字符，以非贪婪模式，表示前接字符的零个或多个匹配。

+ 应用于前接字符，表示前接字符的一个或多个匹配。

+? 应用于前接字符，以非贪婪模式，表示前接字符的一个或多个匹配。

[aeiou] 匹配指定字符集中的一个字符。这里只能是“a”、“e”、 “i”、 “o”或 “u”，不接受其他字符。

[a-z0-9] 使用减号指定字符区间。这里表示一个字符，必须是小写字母或数字。

[^A-Za-z] 第一个字符是^，它表示反向逻辑。这里匹配除了大小写字符之外的其他任意字符。

( )在正则表达式中添加括号，括号内容会丧失匹配功能，但在findall()中可以用于抽取特定部分的字符串，而不是整个字符串。

\b 匹配空字符串，仅用于单词的首尾。

\B 匹配空字符串，但不能用于单词的首尾。

\d 匹配任意十进制数字，等价于[0-9]。

\D 匹配任意非数字字符，等价于0-9。

11.6 Unix用户福利

自20世纪60年以后，Unix操作系统内置了文件搜索的正则表达式功能。它几乎在所有编程语言中通用，只是细节上有所差别。

事实上，Unix内置了一个命令行工具，称为grep（Generalized Regular Expression Parser，通用正则表达式解析器），可以实现本章search()在示例中的相同作用。如果使用Mac或Linux操作系统，你可以在命令行窗口中执行以下语句。

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
![Alt text](./1434462994117.png)
```

这条命令告诉grep，显示mbox-short.txt文件中以“From:”开头的字符串。如果尝试使用grep命令和阅读grep的文档，你会发现Python支持的正则表达式与grep支持的正则表达式存在一些细微差别。例如，grep不支持非空字符“\S”，所以需要使用稍微复杂一点的集合符号“[^]”，表示匹配非空格的任意字符。

11.7 调试

Python包含一些简单明了的内置文档，有助于快速掌握和记忆特定方法的确确切名称。这些文档可以在Python解析器的交互模式下查看。

```
>>> help()

Welcome to Python 2.6!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help> modules
```

如果你知道需要使用的模块名称，使用dir()命令查找这个模块的方法，如下所示：

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

你也可以使用dir命令来查找特定方法的一小部分文档。

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

虽然内置的文档不够详尽，但在急需或没有网络浏览器和搜索引擎的情况下备用。

11.8 术语

脆弱代码：当输入数据是一种特定格式，如果格式上有一点偏差，代码很容易出问题。由于容易出错，这种代码被称为“脆弱代码”。

贪婪匹配：正则表达式中“+”和“*”采用外向扩展，匹配最大可能的字符串。

grep：在大多数Unix操作系统中可以使用的命令，搜索文本文件，通过正则表达式一行行进行匹配。该命令的全称是通用正则表达式解析器（Generalized Regular Expression Parser）。

正则表达式：表达复杂搜索字符串的语言。正则表达式可能包含特定字符串，指明搜索只匹配开头或结尾，以及其他许多类似的功能。

通配符：匹配任意字符的特殊字符。例如，在正则表达式中，句点是一个通配符。

11.9 练习

习题**11.1** 编写一段简单程序，模拟Unix的grep命令操作。要求用户输入一个正则表达式，统计出正则表达式匹配的文本行数。

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4218 lines that matched java$
```

习题**11.2** 编写一个程序，查找“New Revision: 39772”所在的文本行。使用正则表达式的findall()函数抽取每一行中的数字，计算并输出数字的平均值。

```
Enter file:mbox.txt
38549.7949721

Enter file:mbox-short.txt
39756.9259259
```

第12章 网络编程

虽然书中许多示例侧重于读取文件和检索文件中的数据，而如今的互联网上有丰富的信息来源，值得考虑在内。

在本章中，我们伪装成一个网络浏览器，使用超文本传输协议（HyperText Transport Protocol, HTTP）检索网页，读取页面数据并进行解析。

12.1 超文本传输协议 — HTTP

网络协议驱动了整个网络，其本身非常简单。由于Python内置了sockets库，在Python程序中建立网络连接，通过这些套接字检索数据，变得非常容易。

套接字很像文件。不同的是它提供了两个程序之间的双向连接，在一个套接字上可以同时读取和写入。如果你在套接字的一端编写内容，套接字会把数据发送给另一端的应用程序。如果从套接字读取，将得到另一个程序发送的数据。

然而，当套接字另一端没有发送任何数据时，如果你尝试读取套接字，结果就只能等待。如果套接字两端的程序都在等待数据，而不发送任何数据，它们就会这样僵持下去。

程序的一个重要组成部分是与互联网的通讯，即具备某种协议。协议是一组精确规则的集合，决定了谁先谁后，做些什么，对消息如何响应，以及下一步谁来发送等。从某种意义上说，套接字两端的两个程序像是在跳舞，确保不会踩到对方的脚趾。

已有大量文档介绍网络协议。超文本传输协议原文如下：

http://www.w3.org/Protocols/rfc2616/rfc2616.txt

这个文档包含179页，内容复杂详尽。如果你感兴趣，请读完它。如果仅是翻看RFC2616的第36页左右，你会找到GET请求的语法。如果仔细阅读，你会发现是从网络服务器请求文档，与http://www.py4inf.com服务器的80端口建立连接，然后发送表单的一行。

```
GET http://www.py4inf.com/code/romeo.txt HTTP/1.0
```

第二个参数是我们请求的网页，随后我们发送一个空白行。网络服务器将响应文档的一些头部信息和文档内容之后的空白行。

12.2 世界上最简单的网络浏览器

解释HTTP工作原理的最简单方法，也许就是编写一段非常简单的Python程序。与网络服务器建立连接，遵循HTTP协议规则，向服务器请求文档并显示出来。

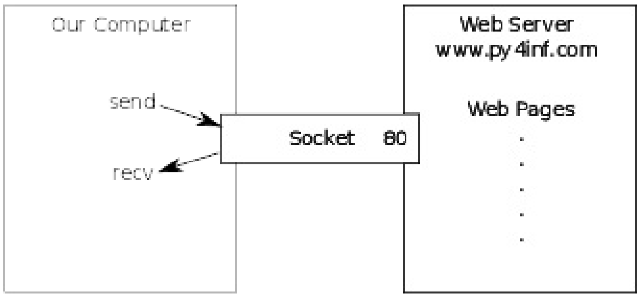
```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('www.py4inf.com', 80))
mysock.send('GET http://www.py4inf.com/code/romeo.txt HTTP/1.0\n\n')

while True:
    data = mysock.recv(512)
    if ( len(data) < 1 ) :
        break
    print data

mysock.close()
```

首先，程序与服务器http://www.py4inf.com在80端口建立一个连接，这个程序扮演了网络浏览器的角色。HTTP协议要求，必须发送GET命令，并在后面跟一个空白行。



发送空白行之后，我们编写一个循环，从套接字中接收512个字符的数据片段，并打印出这些数据，直到没有数据可以读入，即recv()返回一个空字符串。

程序运行结果如下：

```
HTTP/1.1 200 OK
Date: Sun, 14 Mar 2010 23:52:41 GMT
Server: Apache
Last-Modified: Tue, 29 Dec 2009 01:31:22 GMT
ETag: "143c1b33-a7-4b395bea"
Accept-Ranges: bytes
Content-Length: 167
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

程序一开始输出的是网络服务器发送的文档描述的头部信息。例如，Content-Type头部指明文档是一个普通文本文档（text/plain）。

服务器发送了头部信息之后，添加一个空白行表示头部信息发送完毕，然后发送实际的romeo.txt文件数据。

示例展示了如何通过套接字建立低级别的网络连接。套接字用于网络服务器、邮件服务器以及其他许多类型服务器的通讯。找到描述协议的文档，编写代码，根据协议发送和获取数据，要做的就这么多了。

由于最常用的协议是HTTP（即Web）协议，Python针对HTTP协议设计了专门的库来支持网络文档数据的获取。

12.3 通过HTTP检索图像

在以上示例中，我们获取了一个包含换行符的普通文本文件，程序运行时简单地拷贝数据并显示出来。接下来，我们使用HTTP编写一个类似的程序，用来检索图片。在一个字符串中累积数据，截取头部信息，然后将图片数据保存到一个文件中，程序代码如下：

```
import socket
import time

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('www.py4inf.com', 80))
mysock.send('GET http://www.py4inf.com/cover.jpg HTTP/1.0\n\n')


count = 0
picture = "";
while True:
    data = mysock.recv(5120)
    if ( len(data) < 1 ) : break
    # time.sleep(0.25)
    count = count + len(data)
    print len(data),count
    picture = picture + data

mysock.close()

# Look for the end of the header (2 CRLF)
pos = picture.find("\r\n\r\n");
print 'Header length',pos
print picture[:pos]

# Skip past the header and save the picture data
picture = picture[pos+4:]
fhand = open("stuff.jpg","wb")
fhand.write(picture);
fhand.close()
```

程序运行结果如下：

```
$ python urljpeg.py
2920 2920
1460 4380
1460 5840
1460 7300
...
1460 62780
1460 64240
2920 67160
1460 68620
1681 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:15:07 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

针对这个url，Content-Type头部指明文档本身是一个图像（img/jpeg）。程序运行完毕之后，使用图像浏览器打开stuff.jpg文件查看图像数据。

程序运行中调用了recv()函数，每次不会得到5120个字符。调用recv()时，通过网络，我们从网络服务器获得更多字符串。在这个示例中，每一次获得1460或2920个字符，请求上限是5120个字符。

网速不同会导致不同的结果。还要注意的是，最后一次调用recv()，在数据流结束时得到1681个字节，再下一个recv()调用获得零长度的字符串。这就是告诉我们，服务器已经在套接字末尾调用了close()，没有更多数据可发送了。

把time.sleep()前面的注释去掉，这样可以减缓随后的调用。这样一来，每次相隔1/4秒，服务器让我们"靠前"，发送更多的数据。程序的延时间隔执行如下所示：

```
$ python urljpeg.py
1460 1460
5120 6580
5120 11700
...
5120 62900
5120 68020
2281 70301
Header length 240
HTTP/1.1 200 OK
Date: Sat, 02 Nov 2013 02:22:04 GMT
Server: Apache
Last-Modified: Sat, 02 Nov 2013 02:01:26 GMT
ETag: "19c141-111a9-4ea280f8354b8"
Accept-Ranges: bytes
Content-Length: 70057
Connection: close
Content-Type: image/jpeg
```

这次运行中除了第一次和最后一次recv()调用，每次请求新数据都会得到5120个字符。

服务器生成的send()请求与程序生成的recv()请求之间存在一个缓冲区。当程序执行延迟请求时，在某些点上，服务器可能会在套接字中填满缓冲区，并强制暂停，直到程序开始清空缓存区。发送应用或接收应用的暂停行为被称为“流量控制”。

12.4 使用urllib检索网页

与通过HTTP 套接字库手动发送与获取数据相比，Python中有一种更简单的解决方法，使用urllib库。

使用urllib，你可以将网页看成一个文件。只需简单指明需要检索的网页，urllib会处理所有HTPP协议和头部细节。

使用urllib读取romeo.txt文件的代码如下：

```
import urllib

fhand = urllib.urlopen('http://www.py4inf.com/code/romeo.txt')
for line in fhand:
   print line.strip()
```

网页通过urllib.urlopen打开后，我们就可以把它当成一个文件，使用for循环来读取。

程序运行时，我们仅看到文件内容的输出。虽然头部信息仍然会发送，但是urllib代码会处理头部，发送给我们的仅是文件内容。

```
But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

为了演示说明，我们编写一个程序来获取romeo.txt的数据，计算文件中每个单词的频率，代码如下所示：

```
import urllib

counts = dict()
fhand = urllib.urlopen('http://www.py4inf.com/code/romeo.txt')
for line in fhand:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word,0) + 1
print counts
```

同样地，一旦打开了网页，我们就可以把它当做一个本地文件进行读取。

12.5 解析HTML和Web抓取

Python的urllib常见用法之一是网页抓取。网页抓取是编写一个程序，伪装成网络浏览器，检索网页，然后在这些页面中根据模式检视数据。

举例来说，搜索引擎（如Google）会查看网页的源代码，抽取链接到其他页面的超链接，然后检索这些页面，抽取超链接，如此往复下去。使用这种技术，Google爬虫几乎遍历了网络上的所有页面。

Google还使用页面链接的频次来说明一个具体页面的重要性，在搜索结果中表明页面排位的高低。

12.6 使用正则表达式解析HTML

HTML解析的一个简单方法是使用正则表达式进行重复搜索，根据特定模式，抽取出与之匹配的子字符串。

以下是一个简单的网页：

```
<h1>The First Page</h1>
<p>
If you like, you can switch to the
<a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>.
</p>
```

我们构造一个符合语法规则的正则表达式，匹配和抽取以上网页文本中的超链接，正则表达式如下所示：

```
href="http://.+?"
```

这个正则表达式查找以ref="http:// 开头的字符串，之后是一个或多个字符“.+?”，最后是另一个双引号。“.+?”表示以非贪婪方式进行匹配，而不是贪婪方式。非贪婪匹配试图找到最小可能匹配的字符串，贪婪匹配试图找到最大可能匹配的字符串。

正则表达式中的括号表示，我们需要精确匹配的字符串，程序代码如下：

```
import urllib
import re

url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
links = re.findall('href="(http://.*?)"', html)
for link in links:
    print link
```

正则表达式的findall方法返回正则表达式匹配到的字符串列表，仅返回双引号之间的超链接文本。程序运行结果如下：

```
python urlregex.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm

python urlregex.py
Enter - http://www.py4inf.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.py4inf.com/code
http://www.lib.umich.edu/espresso-book-machine
http://www.py4inf.com/py4inf-slides.zip
```

如果HTML网页是良构的和可预测的，正则表达式会处理地很漂亮。但是，由于存在大量“破坏性”的HTML网页，你可能会发现，仅使用正则表达式的解决方案可能会错过一些有效链接或中止于“坏数据”。

强大的HTML解析库可以解决这个问题。

12.7 使用BeautifulSoup解析HTML

已有许多Python库可以帮助你解析HTML与抽取页面中的数据。每个Python库都有优缺点，根据需要进行选择。

举例来看，我们使用BeautifulSoup来简单解析一些HTML输入和抽取链接。从http://www.crummy.com网站下载和安装BeautifulSoup代码。

你可以下载和安装BeautifulSoup，或简单地将BeautifulSoup.py放在你的程序文件夹下。

HTML与XML看起来很像，有一些网页被精心构造为XML。一般而言，大多数HTML会被XML解析器认为是格式不正确而整体拒绝，导致解析失败。BeautifulSoup极大容忍了HTML的缺陷，依然让你能轻易抽取所需的数据。

我们使用urllib读取页面，然后根据锚标签抽取href属性的内容。

```
import urllib
from BeautifulSoup import *

url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
soup = BeautifulSoup(html)

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
   print tag.get('href', None)
```

该程序提示输入一个网址，然后打开网页，读取与传递数据到BeautifulSoup解析器，接下来，检索所有的锚标签，打印出每个标签的href属性内容。

```
python urllinks.py
Enter - http://www.dr-chuck.com/page1.htm
http://www.dr-chuck.com/page2.htm

python urllinks.py
Enter - http://www.py4inf.com/book.htm
http://www.greenteapress.com/thinkpython/thinkpython.html
http://allendowney.com/
http://www.si502.com/
http://www.lib.umich.edu/espresso-book-machine
http://www.py4inf.com/code
http://www.pythonlearn.com/
```

使用BeautifulSoup取出每个标签的不同部分，代码如下：

```
import urllib
from BeautifulSoup import *

url = raw_input('Enter - ')
html = urllib.urlopen(url).read()
soup = BeautifulSoup(html)

# Retrieve all of the anchor tags
tags = soup('a')
for tag in tags:
   # Look at the parts of a tag
   print 'TAG:',tag
   print 'URL:',tag.get('href', None)
   print 'Content:',tag.contents[0]
   print 'Attrs:',tag.attrs
```

程序运行结果如下：

```
python urllink2.py
Enter - http://www.dr-chuck.com/page1.htm
TAG: <a href="http://www.dr-chuck.com/page2.htm">
Second Page</a>
URL: http://www.dr-chuck.com/page2.htm
Content: [u'\nSecond Page']
Attrs: [(u'href', u'http://www.dr-chuck.com/page2.htm')]
```

这些例子仅揭开了BeautifulSoup解析HTML功能的冰山一角。更多内容详见http://www.crummy.com文档与示例。

12.8 使用urllib读取二进制文件

有时，你需要检索非文本（二进制）文件，如图像或视频文件。这些文件的数据直接打印输出是没有用的，但可以通过urllib将URL指向的文件保存在本地硬盘。

该模式打开URL，使用read方法下载整个文档内容，将其存入一个字符串变量（如img），然后将变量内容写入本地文件。程序代码如下：

```
img = urllib.urlopen('http://www.py4inf.com/cover.jpg').read()
fhand = open('cover.jpg', 'w')
fhand.write(img)
fhand.close()
```

该程序通过网络读取所有数据，将它存在计算机内存的img变量中，然后打开文件cover.jpg，将数据写入到硬盘中。如果文件大小小于计算机内存容量，这个程序将会成功运行。

然而，如果是一个大型音频或视频文件，当计算机耗尽内存时，这个程序可能会崩溃或运行极为缓慢。为了避免耗尽内存，我们以区块（或缓冲区）检索数据，在检索下一个区块前将当前区块写入磁盘。这样一来，程序就可以读取任意大小的文件，无需担心耗尽计算机的全部内存。

```
import urllib

img = urllib.urlopen('http://www.py4inf.com/cover.jpg')
fhand = open('cover.jpg', 'w')
size = 0
while True:
    info = img.read(100000)
    if len(info) < 1 : break
    size = size + len(info)
    fhand.write(info)

print size,'characters copied.'
fhand.close()
```

在这个示例中，每次读取100,000个字符，从网络检索下一批100,000个字符数据之前，先将这些字符写入cover.jpg文件。

程序运行结果如下：

```
python curl2.py
568248 characters copied.
```

如果是Unix或Mac计算机，你可以使用操作系统内置命令来执行这个操作：

```
curl -O http://www.py4inf.com/cover.jpg
```

curl命令是"copy URL"的缩写，这两个例子文件命名为curl1.py和curl2.py，可以从http://www.py4inf.com/code下载。它们实现了culr命令相似的功能。curl3.py示例程序以更高效的方式完成二进制文件的读写，这种模式可能对你自己编写程序时有所帮助。

12.9 术语

BeautifulSoup：一个用于HTML文档解析与数据抽取的Python库。它能够处理大多数在浏览器中通常被忽略的，存在缺陷的HTML。BeautifulSoup代码http://从www.crummy.com网站下载。

端口：当与服务器建立套接字连接时，服务器告诉应用程序进行通讯所采用的数字。例如，网络流量通常使用80端口，电子邮件流量使用25端口。

抓取：把程序伪装成网络浏览器，检索网页，查找网页中的内容。通常，程序会根据一个网页中的链接找到下一个网页，实现对网页网络或社交网络的遍历。

套接字：两个应用程序之间的网络连接，彼此可以发送与接收数据。

爬虫：网络搜索引擎的检索页面，然后从该页面所有链接再次发起检索，如此往复下去，直到它们检索到网络上的几乎所有页面。这些页面将用于构建索引，供搜索之用。

12.10 练习

习题**12.1** 修改套接字程序socket1.py，提示用户输入URL，让它可以读取任何网页。你可以使用split('/')拆分URL，抽取出套接字connect调用的主机名。使用try和except增加错误检查，处理用户输入不恰当的网址或不存在的URL这两种情况。

习题**12.2** 修改套接字程序，统计它所接收到的字符数，当显示了3000个字符之后停止显示其他字符。该程序应检索整个文档，统计字符总数并显示在文档结尾。

习题**12.3** 使用urllib重复之前的练习：（1）从URL中检索文档；（2）显示3000个字符；（3）统计文档的字符总数。这里不必担心头部信息，只显示文档内容中前3000个字符即可。

习题**12.4** 修改urlinks.py程序，对检索到的HTML文档抽取和统计段落标签(p)，在程序输出中显示段落数量。不要显示段落文本，仅统计段落总数。在简单网页和复杂网页上测试该程序。

习题**12.5** （进阶）修改socket程序，使其只显示头部和空行之后的检索数据。请记住，recv是按照字符（包括换行及所有）而非行来接收的。

第13章 Web Services

掌握了通过HTTP编写文档检索与解析程序之后，接下来为我们自己创建的文档，开发一种其他应用程序可调用的文档服务，这就不是一件很苦难的事情了。这里说的文档不是显示在网络浏览器中的HTML网页。

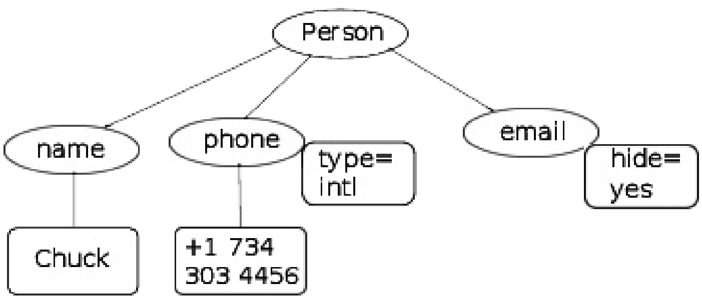
网络数据交换包括两种常见格式：可扩展标记语言XML（eXtensible Markup Language）和JSON（JavaScript Object Notation）。XML已经使用很长时间，适合于交换文档类型的数据。当程序彼此之间只需要交换字典、列表或其他内部信息时，一般会使用JSON。接下来分别介绍这两种格式。

13.1 可扩展标记语言XML

XML与HTML看起非常像，但XML比HTML的结构化程度更高。下面是一个XML文档示例：

```
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>
```

通常可以把XML文档想象成一个树结构，这样有助于理解。示例中顶层标签是person，其他标签是它的子节点，如phone。



13.2 XML解析

下面是一个简单的示例程序，对XML文件进行解析，从中提取一些数据元素：

```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
     +1 734 303 4456
   </phone>
   <email hide="yes"/>
</person>'''

tree = ET.fromstring(data)
print 'Name:',tree.find('name').text
print 'Attr:',tree.find('email').get('hide')
```

调用fromstring将XML的字符串表示转换为一棵XML节点树。当XML被视为一棵树，我们就有一系列方法来抽取XML中的数据片段。

使用find函数对XML树进行搜索，检索出匹配特定标签的节点。每个节点包含一些文本，一些属性（例如隐藏的）以及一些子节点。每个节点都可以成为节点树的顶点。

```
Name: Chuck
Attr: yes
```

XML解析器，例如ElementTree，可以处理许多XML有效性规则，让我们无需操心XML语法规则就可以抽取需要的XML数据片段。由于示例的XML结构过于简单，体现不出这一优势。

13.3 节点循环

通常，XML会拥有多个节点，我们需要写一个循环来处理所有节点。下面的程序使用循环找出user节点：

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
    <users>
        <user x="2">
            <id>001</id>
            <name>Chuck</name>
        </user>
        <user x="7">
            <id>009</id>
            <name>Brent</name>
        </user>
    </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print 'User count:', len(lst)

for item in lst:
    print 'Name', item.find('name').text
    print 'Id', item.find('id').text
    print 'Attribute', item.get('x')
```

使用Python列表来表示XML树中user结构的子树， findall方法检索这个列表。然后，编写一个循环，检索出每一个user节点，打印出name和id的文本元素，以及user节点的x属性值。

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

13.4 JavaScript对象标记 - JSON

JSON格式的灵感来自于JavaScript语言的对象与数组格式。Python的出现早于JavaScript，Python的字典与列表语法对JSON语言有一定影响。所以，JSON格式可视为Python列表与字典的组合。

上面的XML示例使用JSON格式来表示，如下代码所示，两者大致等价：

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
   "email" : {
     "hide" : "yes"
   }
}
```

你会注意到一些差异。首先，XML中“phone”标签有一个“intl”属性，在JSON里处理成键值对。另外，XML的“person”标签没有了，取而代之的是一组大括号。

由于JSON比XML功能少，所以JSON的结构比XML简单。但是，JSON的优势在于，它直接映射成字典与列表的组合。几乎所有编程语言与Python的字典和列表都存在某种程度上的等价关系，所以JSON是两个协作应用程序之间非常合适的数据交换格式。

与XML相比，JSON相对简单，正迅速成为应用程序之间数据交换格式的不二选择。

13.5 JSON解析

通过字典（对象）与列表的嵌套来构造我们需要的JSON。这个示例中用户的列表由键值对集合组成（也就是一个字典）。因此，我们有了一个字典列表。

在下面的程序中，我们使用内置的**json**库来解析JSON，读取其中的数据。仔细比较等价的XML数据和上面的代码，有一点必须提前知晓，JSON的细节较少。我们最终得到的列表包含用户信息，每个用户的信息是键值对集合。JSON的优点是格式简洁，缺点是自描述能力不强。

```
import json

input = "'
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  } ,
  { "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
]"'

info = json.loads(input)
print 'User count:', len(info)

for item in info:
    print 'Name', item['name']
    print 'Id', item['id']
    print 'Attribute', item['x']
```

从解析后的JSON与XML分别抽取数据的两段代码进行比较，你会发现json.loads()得到一个Python列表，通过for循环进行遍历，列表的每个数据项是一个Python字典，其中使用了Python索引操作符来抽取每个用户的各个字节。JSON经过解析后，我们就得到了原生的Python对象与结构。由于返回的数据就是简单的原生Python结构，就没必要使用JSON库继续深入解析JSON了。

程序执行结果如下，与上面的XML版本几乎相同。

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

总之，Web Service的行业发展趋势是从XML转向JSON。由于JSON足够简单，能够直接映射到编程语言已有的原生数据结构，JSON的使用让解析与数据抽取变得更简单。但是，XML比JSON在自我描述方面更强，因此在某些应用程序中XML仍然有一定优势。例如，大多数文字处理器内部存储文档采用XML而不是JSON。

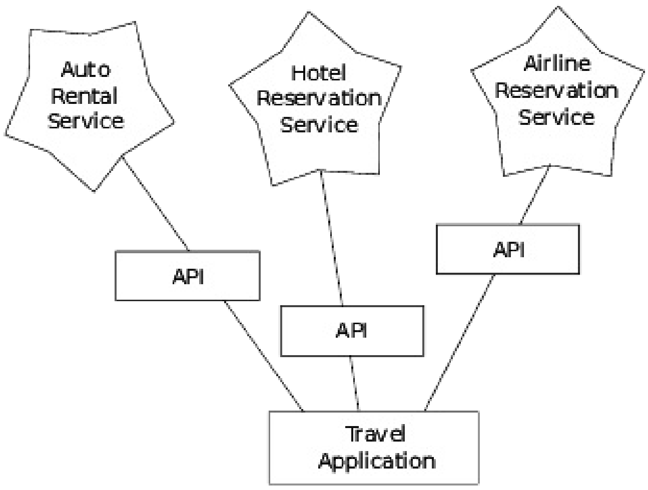
13.6 应用编程接口API

我们已经学习了通过超文本传输协议HTTP在应用程序之间交换数据，了解了使用XML与JSON在这些应用程序之间发送与接收复杂数据的表示方法。

下一步是使用这些技术在应用程序之间定义与签订“合同”。应用程序之间合同的一般名称是应用程序接口或APIs。当我们使用一个API，应用程序通常会提供一组可供其他应用程序使用的服务。应用程序发布的APIs（即“规则”）在访问服务时必须遵守。

当我们开始构建自己的应用程序，功能上要能访问其他应用程序提供的服务，这种方式称为面向服务的架构**SOA**（Service-Oriented Architecture）。SOA方式是应用程序使用其他应用程序服务的总称。非SOA方式是应用程序作为单一独立的程序，自身包含程序运行中所需的所有代码。

互联网应用中存在许多SOA实例。我们可以在一个网站上完成旅行机票与酒店预定以及租车等一系列活动。酒店数据没有存储在航空公司的计算机上。相反，航空公司的计算机与酒店的计算机签订服务合同，检索酒店数据，并呈现给用户。当用户使用航空公司的网站预订了一家酒店，航空公司的网站使用酒店系统的Web Service来完成这笔预订。当涉及整个交易的信用卡支付时，仍然会有其他计算机参与到这个过程。



面向服务的架构拥有许多优点，其中包括：（1）数据始终只存在一处，这对酒店预订这类情况来说特别重要，我们不希望出现多次提交；（2）数据的拥有者能够设置数据的使用规则。基于这些优点，SOA系统必须精心设计，以达到具备良好的性能与满足用户的需求。

应用程序通过网络发布一组可用的API服务，我们称之为Web Services。

13.7 Google地理编码Web Service

Google的Web Service非常优秀，让我们能充分利用其庞大的地理信息数据库。我们可以向Google的地理编码API提交诸如“Ann Arbor, MI”这样的地理搜索字符串。Google会根据搜索字符串返回最佳猜测，在地图上告诉我们想找的地方以及附近的地标性建筑。

地理编码服务是免费的，但有访问次数限制，不能在商业应用程序中无限制使用。如果有一些调查数据，其中被调查者在自由格式的输入框中输入了一个地址，你就可以使用这个API清洗这些数据，效果会很不错。

当使用类似*Google*地理编码*API*的免费*API*时，你需要遵守这些资源的使用规定。如果有太多用户滥用*API*，*Google*会关闭或大幅度缩减这项免费服务。

通过阅读这项服务的在线文档了解使用方法。不过，这个操作非常简单，可以直接在浏览器中测试，在地址栏输入如下URL：

```
http://maps.googleapis.com/maps/api/geocode/json?sensor=false &address=Ann+Arbor%2C+MI
```

粘贴到浏览器之前，确保是原始的URL，移除URL中的任何空格。

下面是一个简单应用程序，提示用户输入一个搜索字符串，调用Google的地理编码API，从返回的JSON中抽取信息。

```
import urllib
import json

serviceurl = 'http://maps.googleapis.com/maps/api/geocode/json?'

while True:
    address = raw_input('Enter location: ')
    if len(address) < 1 : break

    url = serviceurl + urllib.urlencode({'sensor':'false',
          'address': address})
    print 'Retrieving', url
    uh = urllib.urlopen(url)
    data = uh.read()
    print 'Retrieved',len(data),'characters'

    try: js = json.loads(str(data))
    except: js = None
    if 'status' not in js or js['status'] != 'OK':
        print '==== Failure To Retrieve ===='
        print data
        continue

    print json.dumps(js, indent=4)

    lat = js["results"][0]["geometry"]["location"]["lat"]
    lng = js["results"][0]["geometry"]["location"]["lng"]
    print 'lat',lat,'lng',lng
    location = js['results'][0]['formatted_address']
    print location
```

该程序获取输入的搜索字符串，将其作为合适的编码参数，构建URL，使用urllib从Google地理编码API检索文本内容。与固定的网页不同，这里得到的数据取决于发送的参数与Google服务器中存储的地理数据。

一旦获取到JSON数据，我们使用json库对其进行解析，做一些检查以确保收到良好的数据，然后抽取我们需要的数据。

程序执行结果如下（下面只展示了返回的部分JSON数据）：

```
$ python geojson.py
Enter location: Ann Arbor, MI
Retrieving http://maps.googleapis.com/maps/api/
  geocode/json?sensor=false&address=Ann+Arbor%2C+MI
Retrieved 1669 characters
{
    "status": "OK",
    "results": [
        {
            "geometry": {
                "location_type": "APPROXIMATE",
                "location": {
                    "lat": 42.2808256,
                    "lng": -83.7430378
                }
            },
            "address_components": [
                {
                    "long_name": "Ann Arbor",
                    "types": [
                        "locality",
                        "political"
                    ],
                    "short_name": "Ann Arbor"
                }
            ],
            "formatted_address": "Ann Arbor, MI, USA",
            "types": [
                "locality",
                "political"
            ]
        }
    ]
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA
Enter location:
```

你可以下载http://www.py4inf.com/code/geojson.py和http://www.py4inf.com/code/geoxml.py两个文件，搞清楚Google地理编码API的JSON与XML之间的差别。

13.8 安全与API用法

通常情况下，你需要某种类型的API密钥才能访问服务提供者的API。这样设计的初衷是服务提供者想要知道谁在使用他们的服务，以及每个用户的使用情况。可能他们提供免费服务，但会根据服务层次收费，或者在特定时间段限制个体用户的请求数量。

在一些情况中，你一旦得到API密钥，在调用API时只需将密钥作为POST数据的一部分，或者作为URL的一个参数。

在另一些情况中，服务提供者为了增加请求来源的保险性，他们希望你使用共享密钥与密码方式发送加密的签名信息。互联网中签名请求普遍采用OAuth技术。有关OAuth协议详见http://www.oauth.net。

随着Twitter API越来越有价值，Twitter从免费公开的API转向每个API请求需要OAuth签名。值得庆幸的是，许多方便的OAuth库可以免费使用。你可以不必阅读技术规范和从零编写OAuth的实现过程。这些库的复杂性与丰富程度不一。OAuth网站提供各种OAuth库信息。

下面一段示例程序需要从http://www.py4inf.com/code下载三个文件twurl.py、 hidden.py、oauth.py和twitter1.py，把它们放在一个文件夹下。

为了能使用这些程序，需要一个Twitter账号，让你的Python代码作为Twitter的一个应用得以授权，创建key、secret、token与token secret。通过修改hidden.py，将这四个字符串赋予文件中合适的变量。

```
def auth() :
    return { "consumer_key" : "h7L...GNg",
        "consumer_secret" : "dNK...7Q",
        "token_key" : "101...GI",
        "token_secret" : "H0yM...Bo" }
```

Twitter的Web Service通过URL访问，如下所示：
https://api.twitter.com/1.1/statuses/user_timeline.json

所有的安全信息添加完毕之后，完整的URL如下：

```
https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1
```

如果想了解OAuth安全需求的各种参数含义，请阅读OAuth技术规范。

这个程序访问Twitter时，隐藏了文件oauth.py与twurl.py中所有复杂细节。我们只需简单设置hidden.py中的加密信息，然后把请求的URL发送给twurl.augment()函数，库代码会帮我们添加URL需要的所有参数。

程序（twitter1.py）检索了特定Twitter用户的时间线，以JSON格式返回为一个字符串。我们仅打印出字符串前250个字符。

```
import urllib
import twurl

TWITTER_URL='https://api.twitter.com/1.1/statuses/user_timeline.json'

while True:
    print ''
    acct = raw_input('Enter Twitter Account:')
    if ( len(acct) < 1 ) : break
    url = twurl.augment(TWITTER_URL,
        {'screen_name': acct, 'count': '2'} )
    print 'Retrieving', url
    connection = urllib.urlopen(url)
    data = connection.read()
    print data[:250]
    headers = connection.info().dict
    # print headers
    print 'Remaining', headers['x-rate-limit-remaining']
```

程序运行结果如下：

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 17:30:25 +0000 2013","
id":384007200990982144,"id_str":"384007200990982144",
"text":"RT @fixpert: See how the Dutch handle traffic
intersections: http:\/\/t.co\/tIiVWtEhj4\n#brilliant",
"source":"web","truncated":false,"in_rep
Remaining 178

Enter Twitter Account:fixpert
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at":"Sat Sep 28 18:03:56 +0000 2013",
"id":384015634108919808,"id_str":"384015634108919808",
"text":"3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps:\/\/t.co\/2XmHPx7kgX",
"source":"web","truncated":false,
Remaining 177

Enter Twitter Account:
```

Twitter返回时间线数据的同时，还返回了HTTP响应头部中请求的元数据。特殊的头部x-rate-limit-remaining表明，在短时间切断之前我们还能发起的请求数量，这样就可以知道每次API请求中剩余的检索次数。

在下面例子中，我们检索一个用户的Twitter朋友，解析返回的JSON，抽取朋友信息。在解析和4格缩进的“工整打印”之后导出JSON文件。当需要抽取更多字段时，导出的数据可供我们解读。

```
import urllib
import twurl
import json

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

while True:
    print ''
    acct = raw_input('Enter Twitter Account:')
    if ( len(acct) < 1 ) : break
    url = twurl.augment(TWITTER_URL,
        {'screen_name': acct, 'count': '5'} )
    print 'Retrieving', url
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    print 'Remaining', headers['x-rate-limit-remaining']
    js = json.loads(data)
    print json.dumps(js, indent=4)

    for u in js['users'] :
        print u['screen_name']
        s = u['status']['text']
        print '  ',s[:50]
```

由于JSON是Python列表与字典的嵌套集合，我们可以组合使用索引操作与for循环操作，使用很少的Python代码遍历返回的数据结构。

程序运行结果如下（为方便页面显示，一些数据被压缩）：

```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14
{
    "next_cursor": 1444171224491980205,
    "users": [
        {
            "id": 662433,
            "followers_count": 28725,
            "status": {
                "text": "@jazzychad I just bought one .__.",
                "created_at": "Fri Sep 20 08:36:34 +0000 2013",
                "retweeted": false,
            },
            "location": "San Francisco, California",
            "screen_name": "leahculver",
            "name": "Leah Culver",
        },
        {
            "id": 40426722,
            "followers_count": 2635,
            "status": {
                "text": "RT @WSJ: Big employers like Google ...",
                "created_at": "Sat Sep 28 19:36:37 +0000 2013",
            },
            "location": "Victoria Canada",
            "screen_name": "_valeriei",
            "name": "Valerie Irvine",
    ],
    "next_cursor_str": "1444171224491980205"
}
leahculver
   @jazzychad I just bought one .__.
_valeriei
   RT @WSJ: Big employers like Google, AT&amp;T are h
ericbollens
   RT @lukew: sneak peek: my LONG take on the good &a
halherzog
   Learning Objects is 10. We had a cake with the LO,
scweeker
   @DeviceLabDC love it! Now where so I get that "etc

Enter Twitter Account:
```

在输出的最后，我们看到for循环读取了Twitter账号drchuck的5位新近朋友，打印出每位朋友的最近状态。返回的JSON中还有更多数据可用。此外，如果仔细查看程序输出，你会发现，特定账号的“找到他的朋友”与时间线查询在一个时间段的请求数量有不同的访问限制。

这些安全的API密钥让Twitter有充分信心认为，他们知道谁在使用他们的API以及数据使用情况。访问限制让我们可以做一些简单的个人数据检索，但不能用于构建每天有百万级API数据访问的产品。

13.9 术语

API：应用程序接口 —— 应用程序之间的合同，定义了两个应用组件之间交互的模式。

ElementTree：用于解析XML数据的Python内置库。

JSON：JavaScript Object Notation —— 基于JavaScript对象语法的结构化数据标识格式。

REST：表述性状态转移 —— 一种Web Service风格，通过HTTP协议提供应用程序间的资源访问。

SOA：面向服务的架构 —— 应用程序由跨网络连接的组件组成。

XML：可扩展标识语言 —— 结构化数据标识格式。

13.10 练习

习题**13.1** 修改http://www.py4inf.com/code/geojson.py或http://www.py4inf.com/code/geoxml.py文件，从检索到的数据中打印出2位字符的国家代码。添加错误检查，确保国家代码不存在时的异常处理。当程序正常工作了，搜索“Atlantic Ocean”，让程序可以处理不属于任何国家的地理位置。

第14章 数据库与结构化查询语言SQL

14.1 什么是数据库

数据库是经过组织的、存储数据的文件。从这个意义上讲，大多数数据库的组织方式与字典类似，数据库实现键与值之间的映射。数据库与字典的最大区别在于，数据库存储在磁盘（或其他永久存储器）上，程序运行结束后数据会永久存在。正是由于数据库存储在永久存储上，它能存储的数据远远多于字典。字典受到计算机内存大小的限制。

与字典类似，数据库软件被设计为快速保留插入的与访问的数据，即使是大量数据的情况亦如此。数据库软件通过对添加的数据构建索引来维护性能，让计算机可以快速跳转到特定数据项。

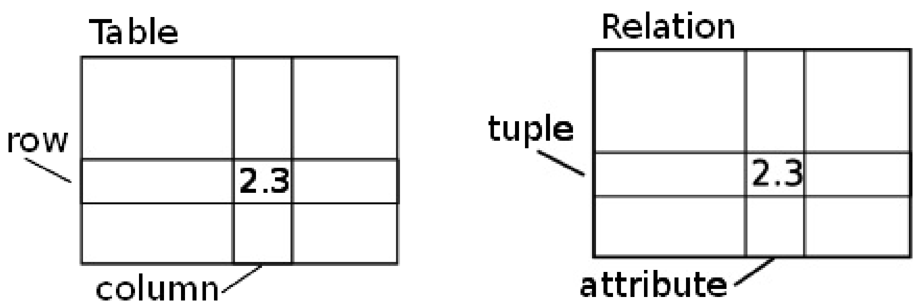
广泛应用的数据库系统包括Oracle、MySQL、Microsoft SQL Server、 PostgreSQL和SQLite。本书关注SQLite，因为它是一个非常通用的数据库，而且已经内建在Python。SQLite被设计为嵌入到其他应用程序，提供应用程序内的数据库支持。例如，Firefox浏览器把SQLite数据库作为内部使用，其他很多产品也这样做。

http://sqlite.org/

SQLite非常适合信息科学中的一些数据处理问题，比如本章介绍的Twitter爬虫应用。

14.2 数据库概念

初次接触数据库，可将其视为多个工作表的电子表格。数据库的主要数据结构包括表、行与列。



在关系型数据库中，表、行与列的专业定义为关系、元组与属性。本章将使用非专业化术语。

14.3 SQLite管理器（Firefox插件）

本章重点使用Python对SQLite数据库文件进行操作，许多操作可以用SQLite数据库管理器（一个Firefox插件）更方便地完成。免费下载地址如下：

https://addons.mozilla.org/en-us/firefox/addon/sqlite-manager/

使用Firefox浏览器可以在数据库中轻松创建表、插入数据、编辑数据、以及执行简单的SQL查询。

从某种意义上讲，在文本文件的处理方面，数据库管理器与文本编辑器类似。如果要对文本文件进行少量修改操作，你可以在文本编辑器中打开它，并根据需要修改。如果要对文本文件进行大量修改时，通常需要编写一个简单的Python程序。类似的，数据库同样存在相同的模式。在数据库管理器中执行一些简单操作，在Python中可以方便地处理一些复杂操作。

14.4 创建一张数据库的表

与Python的列表与字典相比，数据库需要更多的结构定义[1]。

创建一个数据库的表，我们必须根据每一列存储的数据情况，预先在数据库中定义表的每一列名称和数据类型。数据库软件知道了每一列的数据类型，根据特定数据类型，它可以选择最有效的数据存储与检索方法。

以下网址介绍了SQLite支持的各种数据类型：

http://www.sqlite.org/datatypes.html

一开始就定义好数据结构可能不是很方便，但是这样做的好处是，当数据库包含大量数据可以提供快速的数据访问。

以下代码创建了一个数据库文件和带有两列的Tracks表。

```
import sqlite3

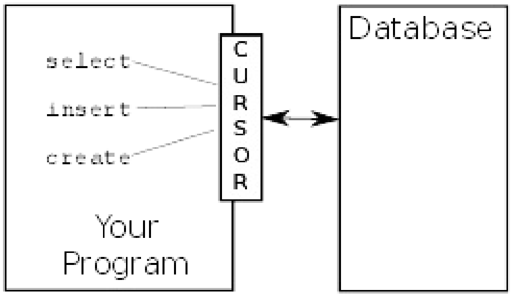
conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Tracks ')
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')

conn.close()
```

connect操作建立了与当前目录中music.sqlite3数据库文件的“连接”。如果文件不存在，则创建它。之所以称为“连接”，因为数据库有时存储在单独的数据库服务器上，与我们运行的应用程序不在同一个服务器上。在这个简单示例中，数据库作为一个本地文件，与Python代码处在同一个目录下。

游标（cursor）类似一个文件句柄，可以对数据库中的数据执行操作。当处理文本文件时，cursor()的调用与open()方法相似。



当有个游标，我们使用execute()方法，开始对数据库的内容执行命令。

数据库命令使用专门的语言，在众多数据库厂商中间已经标准化，用户只需学习一种数据库语言即可。数据库语言称为结构化查询语言，简称为SQL。

http://en.wikipedia.org/wiki/SQL

在这个例子中，我们对数据库执行两条SQL命令。按照惯例，我们用大写显示SQL关键词，其他部分如表和列名显示为小写。

如果Track表已经存在，第一条SQL命令就移除Tracks表。这一做法可以让我们反复执行相同的程序来创建Tracks表，而不会导致不错。需要注意的是，DROP TABLE命令会删除表以及数据库中表的所有内容，也就是说没有撤销的可能。

```
cur.execute('DROP TABLE IF EXISTS Tracks ')
```

第二条命令创建Tracks表，包括文本型的title列与整数型的plays列。

```
cur.execute('CREATE TABLE Tracks (title TEXT, plays INTEGER)')
```

现在，我们已经创建好Tracks表，接下来使用SQL的INSERT操作，向表中添加一些数据。我们再次与数据库建立连接，获得游标（cursor）。通过游标执行SQL命令。

SQL的INSERT命令表明所使用的表，通过列举字段来定义新列，(title, plays)后面跟VALUES具体的列值，从而产生一个新行。我们指定值为(?, ?)，这表示实际值通过第二个参数的一个元组('My Way', 15)来传递，最后调用execute()方法。

```
import sqlite3

conn = sqlite3.connect('music.sqlite3')
cur = conn.cursor()

cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
    ( 'Thunderstruck', 20 ) )
cur.execute('INSERT INTO Tracks (title, plays) VALUES ( ?, ? )',
    ( 'My Way', 15 ) )
conn.commit()

print 'Tracks:'
cur.execute('SELECT title, plays FROM Tracks')
for row in cur :
   print row

cur.execute('DELETE FROM Tracks WHERE plays < 100')
conn.commit()

cur.close()
```

首先，我们向表中插入两行，使用commit()提交命令将数据写入数据库文件。

Tracks

title	plays
Thunderstruck	20
My Way	15

然后，我们用SELECT命令检索刚插入表中的行。SELECT命令首先指定（title, plays）列，之后是数据检索的来源表。执行SELECT语句后，游标可以让我们用for语句进行循环。为了提高效率，当执行SELECT语句时，游标并不会从数据库中读取所有数据。相反，数据是在for循环时按需读取。

程序运行结果如下：

```
Tracks:
(u'Thunderstruck', 20)
(u'My Way', 15)
```

for循环找到两行，每一行是一个Python元组，其中第一个值是歌曲名称（title），第二个值是播放次数（plays）。不用担心，title字符串以u’开头。这说明字符串使用Unicode，即能够存储非拉丁字符集。

在程序末尾，我们执行SQL的DELETE命令，删除刚才创建的行，以便可以反复运行这个程序。DELETE命令使用了WHERE子句，用来表达一个选择条件，这样SQL命令在数据库中只对条件匹配的行进行操作。在本示例中，条件应用于所有行，因此我们可以清空表，反复执

行程序。在DELETE命令执行后，使用commit()提交命令将数据从数据库中删除。

14.5 结构化查询语言SQL小结

至此，我们在Python示例中使用了结构化查询语言，介绍了一些SQL命令的基本知识。本节专门介绍SQL语言，简要介绍SQL语法。

虽然数据库行业中存在众多数据库厂商，但结构化查询语言SQL的标准化使得不同厂商之间的数据库系统可以进行数据互通与移植。

关系型数据库由表、行与列组成。列的常见字段类型包括文本、数值与日期数据。当创建表时，需要指明列的名称与字段类型：

```
CREATE TABLE Tracks (title TEXT, plays INTEGER)
```

使用SQL的INSERT命令向表中插入一行：

```
INSERT INTO Tracks (title, plays) VALUES ('My Way', 15)
```

INSERT语句指定表的名称，之后是想要插入新行的字段（列）的列表，然后是VALUES关键词及其后面每个字段对应的值列表。

SQL的SELECT命令从数据库中检索行与列。SELECT语句指定想要检索的列，WHERE子句用于筛选出符合条件的行。另外，可选的ORDER BY子句控制返回的行的显示顺序。

```
SELECT * FROM Tracks WHERE title = 'My Way'
```

*星号表示从数据库返回WHERE子句匹配到的行的所有列。

请注意，与Python不同的是，SQL的WHERE子句使用一个等号表示相等，而不是两个等号。WHERE子句的其他逻辑操作符包括<、>、 <=、 >=和!=，以及AND、OR与括号，这些可用于编写逻辑表达式。

根据一个字段对返回的行进行排序的查询如下：

```
SELECT title,plays FROM Tracks ORDER BY title
```

要移除行，需要在SQL的DELETE语句增加一个WHERE子句。WHERE子句决定哪些行可被删除：

```
DELETE FROM Tracks WHERE title = 'My Way'
```

在一个表中可以用SQL的UPDATE语句对一行或多行的一个列或多列进行更新。

```
UPDATE Tracks SET plays = 16 WHERE title = 'My Way'
```

UPDATE语句先指明待更新的表，在SET关键词之后设置修改的字段及其取值，然后可以用WHERE子句（可选的）选择要更新的行。一个UPDATE语句会修改WHERE子句匹配到的所有行，若不指定WHERE子句，它将更新表中所有的行。

以上是数据创建与维护的四个基本SQL命令（INSERT、 SELECT、 UPDATE和DELETE）。

14.6 使用数据库爬取Twitter

在本节中，我们编写一个简单的爬虫程序，通过Twitter账号采集数据，然后建立数据库。请注意：谨慎执行这个程序，不要抓取太多数据或长时间执行程序，这样会导致你的*Twitter*账号被封。

任何类型的爬虫程序都面临一个问题，它需要能被停止和重启多次，你也不想丢失已获取到的数据。你不希望总是在一开始重启数据检索，因此把检索到的数据存储起来，让爬虫程序能启动备份，并在它离开的地方继续检索。

我们通过检索一个用户的Twitter朋友及他们的状态，循环朋友列表，向数据库添加每个朋友的信息，以备后续检索。当处理了一个用户的Twitter朋友，我们登录数据库，检索朋友中的一个。重复这个操作，挑选一个“未访问过的”用户，检索他的用户列表，添加列表中没有的朋友，以备下次访问。

我们也追踪数据库中特定朋友的出现次数，以此查看“人气”情况。

通过存储已知账号的列表，不论是否检索这个账号，数据库中账号的人气情况已经存储在计算机磁盘，这样停止或重启程序多少次都没关系。

这个程序有些复杂，它基于前面的Twitter API程序代码。

Twitter爬虫程序源代码如下：

```
import urllib
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()
```

```
cur.execute('''
CREATE TABLE IF NOT EXISTS Twitter
(name TEXT, retrieved INTEGER, friends INTEGER)''')

while True:
    acct = raw_input('Enter a Twitter account, or quit: ')
    if ( acct == 'quit' ) : break
    if ( len(acct) < 1 ) :
        cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
            print 'No unretrieved Twitter accounts found'
            continue

    url = twurl.augment(TWITTER_URL,
               {'screen_name': acct, 'count': '20'} )
    print 'Retrieving', url
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    # print 'Remaining', headers['x-rate-limit-remaining']
    js = json.loads(data)
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )

    countnew = 0
    countold = 0
    for u in js['users'] :
        friend = u['screen_name']
        print friend
        cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
                (count+1, friend) )
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (name, retrieved, friends)
                VALUES ( ?, 0, 1 )''', ( friend, ) )
            countnew = countnew + 1
    print 'New accounts=',countnew,' revisited=',countold
    conn.commit()

cur.close()
```

数据库存在于spider.sqlite3文件中，包括一个Twitter表。Twitter表的每一行包括账号名、是否检索过这个账号的朋友以及这个账号被加好友的次数。

在程序的主循环中，提示用户输入一个Twitter账号名或退出程序。如果用户输入一个Twitter账号，程序就检索朋友列表和用户状态，如果数据库中没有这个朋友，则添加进去。如果该朋友已经存在于列表中，我们对friends字段值加一。

当用户按下回车键，在数据库中寻找下一个还未检索过的Twitter账号，检索该账号的朋友与状态，把添加他们到数据库，或更新它们，增加friends字段的统计值。

当获取到朋友列表与状态，我们对返回的JSON中所有的user数据项进行循环，检索每个用户的screen_name。然后，使用SELECT语句检查screen_name是否已经存储到数据库中了。如果记录存在的话，检索朋友数（friends字段）。

```
countnew = 0
  countold = 0
  for u in js['users'] :
      friend = u['screen_name']
      print friend
      cur.execute('SELECT friends FROM Twitter WHERE name = ? LIMIT 1',
          (friend, ) )
      try:
          count = cur.fetchone()[0]
          cur.execute('UPDATE Twitter SET friends = ? WHERE name = ?',
              (count+1, friend) )
          countold = countold + 1
      except:
          cur.execute("'INSERT INTO Twitter (name, retrieved, friends)
              VALUES ( ?, 0, 1 )"', ( friend, ) )
          countnew = countnew + 1
  print 'New accounts=',countnew,' revisited=',countold
  conn.commit()
```

当游标执行SELECT语句，我们必须检索表的行。用for语句来实现，由于只检索了一行（LIMIT 1）,我们使用fetchone()方法获取第一（也是唯一）行，这就是SELECT操作的结果。由于fetchone()以元组返回行，即使仅有一个字段也是如此。我们用[0]取出元组的第一个值，得到变量count的当前朋友数。

如果获取成功，我们使用SQL的UPDATE语句和WHERE子句，对匹配到的朋友账号所在行的friends列值加一。请注意，SQL语句中有两个占位符（即问号）， execute()的第二个参数是两元素元组，其中的值会替换SQL的占位符。

如果try区块的代码失效，可能是因为SELECT语句的WHERE name = ?子句没有匹配到记录。在except区块，我们使用SQL的INSERT语句，向表中添加朋友的screen_name，另外一个指示符表示我们还没有获取到screen_name，这时将朋友数设为0。

第一次执行程序，输入一个Twitter账号，程序运行结果如下：

```
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit: quit
```

由于是第一次执行这个程序，数据库是空的，我们创建了一个数据库文件spider.sqlite3，向数据库添加一张Twitter表。然后获取一些朋友，将他们的信息存储到之前空的数据库中。

此时，我们想要编写一个简单的数据库导出程序，用来查看spider.sqlites3文件：

```
import sqlite3

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur :
   print row
   count = count + 1
print count, 'rows.'
cur.close()
```

如果再次执行Twitter爬虫程序，程序运行结果如下：

```
(u'opencontent', 0, 1)
(u'lhawthorn', 0, 1)
(u'steve_coppin', 0, 1)
(u'davidkocher', 0, 1)
(u'hrheingold', 0, 1)
...
20 rows.
```

我们看到每个screen_name有一行，没有获取该字段本身的数据，数据库中每人有一个朋友。

现在，在数据库里可以看到第一个Twitter账号（drchuck）的朋友已经获取到。我们再次执行这个程序，只需按下回车键，不用再输入Twitter账号，程序就会检索下一个“未处理账号”的朋友信息。

```
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 18  revisited= 2
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

由于我们按下了回车键（即没有指定Twitter账号），执行下面的代码：

```
if ( len(acct) < 1 ) :
    cur.execute('SELECT name FROM Twitter WHERE retrieved = 0 LIMIT 1')
    try:
        acct = cur.fetchone()[0]
    except:
        print 'No unretrieved twitter accounts found'
        continue
```

我们使用SQL的SELECT语句获取第一个用户的名称（LIMIT 1），但该用户的“是否访问过”字段值还是0。我们还在try/except区块中使用fetchone()[0]模式，从检索到的数据中抽取screen_name，或是得到一个错误消息和循环备份。

如果成功获取到一个未处理的screen_name，检索该账户数据的程序代码如下：

```
url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'} )
print 'Retrieving', url
connection = urllib.urlopen(url)
data = connection.read()
js = json.loads(data)

cur.execute('UPDATE Twitter SET retrieved=1 WHERE name = ?', (acct, ) )
```

一旦成功获取数据，我们使用UPDATE语句设置retrieved列值为1，表示已经完成对该账号的朋友检索。这保证了不会重复检索相同的数据，处理继续进行，最终形成Twitter朋友网络。

如果我们执行friend程序，按两次回车键，检索下一个未被访问的朋友的朋友，然后执行dumping程序，程序输出结果如下：

```
(u'opencontent', 1, 1)
(u'lhawthorn', 1, 1)
(u'steve_coppin', 0, 1)
(u'davidkocher', 0, 1)
(u'hrheingold', 0, 1)
...
(u'cnxorg', 0, 2)
(u'knoop', 0, 1)
(u'kthanos', 0, 2)
(u'LectureTools', 0, 1)
...
55 rows.
```

由此可见，我们正确记录了已经访问过的lhawthorn和opencontent两个账号的信息。另外，cnxorg和kthanos已经有了粉丝。由于已经检索了三个用户（drchuck、opencontent与lhawthorn）的朋友，表中已有55行。

每次执行程序与按下回车键时，它会选择下一个未访问的账号（这里的下一个账号是steve_coppin），获取他们的朋友，标记他们，循环steve_coppin的每一位朋友，将他们添加到数据库。如果他们已经存在于数据库，更新他们的朋友数。

由于程序的数据全部存储在数据库的磁盘上，爬虫活动可以被任意多次暂停或继续，数据都不会丢失。

14.7 基础数据建模

关系型数据库的真正实力在于，创建多个表以及表间连接。将应用数据分解为多个表并确立两个表间的关系，这一过程称为数据建模。显示表与表间关系的设计文档称为数据模型。

数据建模是相对复杂的技能，本节仅介绍最基础的关系型数据建模。数据建模的更多细节，详见以下维基页面：

http://en.wikipedia.org/wiki/Relational_model

让我们来看Twitter爬虫程序，不仅统计一个用户的朋友数，我们希望得到所有的入链关系，即找到特定账号的所有粉丝的列表。

由于每个人都可能会有许多粉丝，所以不能简单添加一列到Twitter表。因此，我们新建一个表来跟踪朋友对。下面是新建表的一种方法：

```
CREATE TABLE Pals (from_friend TEXT, to_friend TEXT)
```

每当遇到drchuck的一个粉丝，我们向表中插入一行：

```
INSERT INTO Pals (from_friend,to_friend) VALUES ('drchuck', 'lhawthorn')
```

当处理了drchuck的20个朋友的Twitter消息源（feed），我们插入“drchuck”作为第一参数的20条记录，这个字符串在数据库重复出现了多次。

重复的字符串数据破坏了数据库规范化的最佳实践。数据库规范化指相同的字符串数据在数据库只能存在一处。如果需要数据出现多次，要为数据创建一个数字键，通过该键引用实际的数据。

在实际应用中，字符串比整数在计算机磁盘与内存上占用更多空间，处理器也需要更多时间进行比较和排序。如果仅有几百条数据，那么存储与处理器耗时并没什么问题。但当数据库中包括百万用户，以及可能的一亿朋友链接，尽可能快速扫描数据就显得非常重要了。

我们把Twitter账号存储在People表，而不是之前示例的Twitter表。People表用额外的一列来存储与该Twitter用户的行数相关的数值键。SQLite的INTEGER PRIMARY KEY这一特殊的数据列类型能为插入的任一行自动增加键值。

新建People表，包括一个额外的id列：

```
CREATE TABLE People
(id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)
```

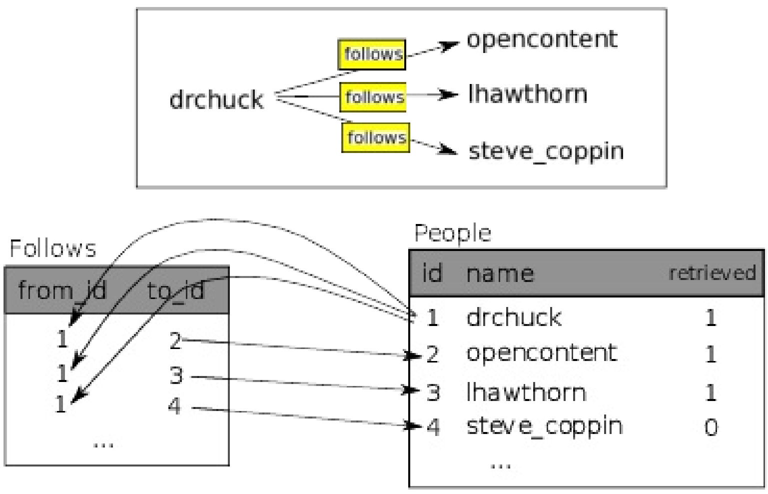
请注意，我们不再维护People表每一行的朋友数。当选择INTEGER PRIMARY KEY作为id列的字段类型，这表明我们希望SQLite来管理该列，在插入一行时自动赋予唯一的数值键。我们还添加了关键词UNIQUE，表示不允许SQLite为两个行插入相同的值。

与之前创建的Pals表不同，我们创建了一个Follows表，包括from_id和to_id两个整数列，以及一个表级约束，表中from_id和to_id必须唯一，即不能在数据库中插入重复的行。

```
CREATE TABLE Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id) )
```

向表中添加UNIQUE子句，当试图插入记录时，我们要求数据库强制执行这一套规则。在程序中创建这些规则的方便性稍后会说明。这些规则避免我们犯错误，并简化了一些代码编写。

从本质上说，Follows表的创建实际是构建了一个“关系”，一个用户是其他人的粉丝，将其表示成一个数值对，代表与他联系的用户，以及关系的方向。



14.8 多表编程

我们使用之前的两个表、主键和键引用来重做Twitter爬虫程序。新版本的程序代码如下：

```
import urllib
import twurl
import json
import sqlite3

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'
```

```
conn = sqlite3.connect('friends.sqlitesqlite3')
cur = conn.cursor()

cur.execute("'CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)"')
cur.execute("'CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))"')

while True:
    acct = raw_input('Enter a Twitter account, or quit: ')
    if ( acct == 'quit' ) : break
    if ( len(acct) < 1 ) :
        cur.execute("'SELECT id, name FROM People
            WHERE retrieved = 0 LIMIT 1"')
        try:
            (id, acct) = cur.fetchone()
        except:
            print 'No unretrieved Twitter accounts found'
            continue
    else:
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (acct, ) )
        try:
            id = cur.fetchone()[0]
        except:
            cur.execute("'INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)"', ( acct, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',acct
                continue
            id = cur.lastrowid

    url = twurl.augment(TWITTER_URL,
       {'screen_name': acct, 'count': '20'} )
    print 'Retrieving account', acct
    connection = urllib.urlopen(url)
    data = connection.read()
    headers = connection.info().dict
    print 'Remaining', headers['x-rate-limit-remaining']

    js = json.loads(data)
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE People SET retrieved=1 WHERE name = ?', (acct, ) )

    countnew = 0
    countold = 0
    for u in js['users'] :
        friend = u['screen_name']
        print friend
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (friend, ) )
```

```
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( friend, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',friend
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Follows (from_id, to_id)
            VALUES (?, ?)''', (id, friend_id) )
    print 'New accounts=',countnew,' revisited=',countold
    conn.commit()

cur.close()
```

这个程序变得有些复杂了，介绍了通过整数键连接表格的使用模式。基本模式如下：

1. 创建带有主键与约束的表。
2. 当一个用户（即账号名称）拥有一个逻辑键，我们需要用户的id值。根据People表中是否有该用户，（1）查找People表中的用户，获取该用户的id值，或（2）向People表添加用户，为新增行添加id值。
3. 插入一行，表示“粉丝”关系。

以下依次介绍每一个步骤。

14.8.1 数据库表约束

设计表结构时，我们告诉数据库系统强制执行一些规则。这些规则帮助我们避免出错，不要把错误的数据写入表中。创建表的代码如下：

```
cur.execute('''CREATE TABLE IF NOT EXISTS People
    (id INTEGER PRIMARY KEY, name TEXT UNIQUE, retrieved INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Follows
    (from_id INTEGER, to_id INTEGER, UNIQUE(from_id, to_id))''')
```

我们定义People表中的name列必须是唯一的（UNIQUE）。同时，定义Follows表每一行两个数字的组合必须唯一。这些约束避免了多次添加同一个关系。

以下代码体现了这些约束的优势：

```
cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
    VALUES ( ?, 0)''', ( friend, ) )
```

我们在INSERT语句中添加OR IGNORE子句，这表示如果有一个INSERT违反了"name必须唯一"的规则，那么数据库将忽略这个INSERT。数据库约束作为一个安全网络，确保我们不会在无意中犯错。

同样地，以下代码确保不会重复添加同一个Follows关系。

```
cur.execute('''INSERT OR IGNORE INTO Follows
    (from_id, to_id) VALUES (?, ?)''', (id, friend_id) )
```

同样地，如果违反了Follows行的唯一性约束，只需告诉数据库忽略INSERT即可。

14.8.2 检索与插入一条记录

当提示用户输入一个Twitter账号，如果账号已存在，我们必须找到它的id值。如果People表中还没有该账号，我们必须插入一条记录，并得到该插入行的id值。

这是一个很常见的模式，在前面的程序中用到过2次。当我们从已获取的Twitter的JSON数据中获取user节点的screen_name，本节代码演示了如何检索一个朋友账号的id。

随着数据的累积，用户账号可能已经存在于数据库中。我们需要先使用SELECT语句，检查People表中该账号是或否存在。

如果try部分一切进展顺利[2]，我们使用fetchone()获取该记录，然后检索返回的元组的第一个（也是唯一）元素，将其存储为friend_id。

如果SELECT执行失败，fetchone()[0]也会失败，然后控制跳转到except部分。

```
friend = u['screen_name']
        cur.execute('SELECT id FROM People WHERE name = ? LIMIT 1',
            (friend, ) )
        try:
            friend_id = cur.fetchone()[0]
            countold = countold + 1
        except:
            cur.execute('''INSERT OR IGNORE INTO People (name, retrieved)
                VALUES ( ?, 0)''', ( friend, ) )
            conn.commit()
            if cur.rowcount != 1 :
                print 'Error inserting account:',friend
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
```

如果以except代码结束，这意味着，没有发现记录，必须插入新行。我们使用INSERT OR IGNORE仅是避免出错，然后调用commit()来强制数据库对提交进行更新。当写入完成后，我们通过cur.rowcount检查有多少行受到影响。由于我们尝试插入一个单行，如果受影响行的数

字不是1，那么将导致错误。

如果INSERT执行成功，我们通过cur.lastrowid找出数据库为新建行赋予的id列值。

14.8.3 存储朋友关系

一旦知道了JSON数据中Twitter用户与朋友的键值，在Follows表中插入两个值就是件简单的事情了，程序代码如下：

```
cur.execute('INSERT OR IGNORE INTO Follows (from_id, to_id) VALUES (?, ?)',
    (id, friend_id) )
```

请注意，根据表格创建时的唯一性约束，避免了重复插入同一个关系，然后在INSERT语句中添加OR IGNORE。

程序运行结果如下所示：

```
Enter a Twitter account, or quit:
No unretrieved Twitter accounts found
Enter a Twitter account, or quit: drchuck
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 20  revisited= 0
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit:
Retrieving http://api.twitter.com/1.1/friends ...
New accounts= 17  revisited= 3
Enter a Twitter account, or quit: quit
```

从drchuck账号开始，让程序自动获取下两个账号，并添加到数据库。

当程序执行完成后，以下是People和Follows表的头几行：

```
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
```

可以看出，People表中的id、name与visited字段，关系Follows表结尾的数字。在People表中，我们看到前三个用户已经被访问过，他们的数据已被获取。Follows表的数据表明，drchuck（用户1）是所显示前五行的用户的朋友。这是由于我们获取和存储的第一个数据是drchuck的Twitter朋友。如果打印出Follows表的更多行，就会看到用户2和用户3的朋友。

14.9 键的三种类型

现在我们已经开始构建数据模型了，将数据放入多个关联表中，使用键来连接这些表中的行。我们需要知道一些有关键的术语。数据库模型中一般存在三种类型的键。

- 逻辑键是“真实世界”中可以检索行的键。在示例数据模型中，name字段是一个逻辑键。它是用户的屏幕名称，在程序中通过name字段多次检索用户的行。你会发现，对一个逻辑键添加UNIQUE约束，这是有道理的。由于逻辑键是我们从“外部世界”如何检索表中的一行，允许表中存在相同值的多行没有多大意义。
- 主键通常是由数据库自动赋予的一个数字。它对外部程序而言没有意义，仅用于把来自不同表的行连接在一起。当我们检索表中的行，通常搜索主键是最快的。由于主键是整数值，占用极少的存储空间，能够快速进行比较与排序。在示例数据模型中，id字段是主键。
- 外键通常是指向不同表中相关行的主键的一个数值。示例数据模型中的外键是from_id。

我们使用一些命名惯例，比如主键名为id，那么将_id后缀添加到对应外键的名称中。

14.10 使用JSON获取数据

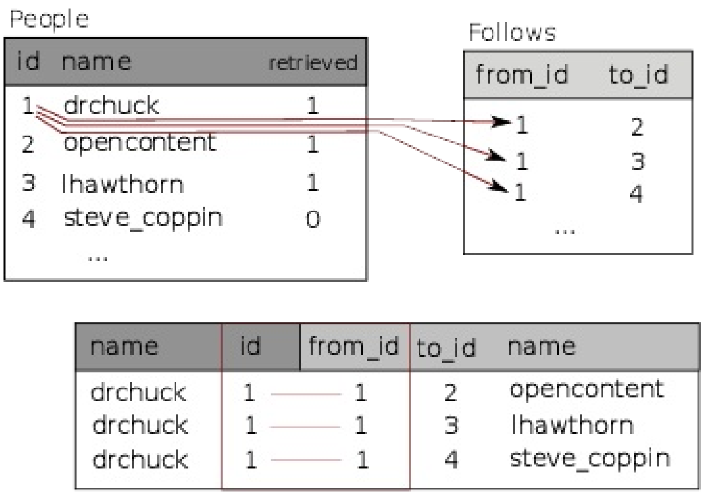
我们已经了解了数据库规范化原则，将数据分成两个表，通过主键和外键将两个表连接起来，然后通过SELECT将跨表格的数据组装在一起。

SQL使用连接（JOIN）子句把表重新连接。在JOIN子句中可以指定用以连接表之间行的字段。

下面是带有JOIN子句的SELECT语句示例：

```
SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 1
```

JOIN子句表示，从Follows与People两个表中选择所有字段。ON子句表示，两个表怎样被连接在一起。选取People表的行，然后将Follows表的from_id字段与People表的id字段值相同的行附加在后面。



连接的结果是创建了一个相当长的“元行”（meta-rows），包括People表的字段与Follows表中匹配的字段。由于People表的id字段与Follows表中的from_id字段之间存在多个匹配，JOIN会为每一个匹配到的行创建一个元行，根据需要重复数据。

多表数据库驱动的Twitter爬虫程序多次执行的代码如下：

```
import sqlite3

conn = sqlite3.connect('spider.sqlite3')
cur = conn.cursor()

cur.execute('SELECT * FROM People')
count = 0
print 'People:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.execute('SELECT * FROM Follows')
count = 0
print 'Follows:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.execute('''SELECT * FROM Follows JOIN People
    ON Follows.from_id = People.id WHERE People.id = 2''')
count = 0
print 'Connections for id=2:'
for row in cur :
   if count < 5: print row
   count = count + 1
print count, 'rows.'

cur.close()
```

在这个程序中，我们首先整体导出People与Follows表，然后导出连接表的数据子集。

程序输出结果如下：

```
python twjoin.py
People:
(1, u'drchuck', 1)
(2, u'opencontent', 1)
(3, u'lhawthorn', 1)
(4, u'steve_coppin', 0)
(5, u'davidkocher', 0)
55 rows.
Follows:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
60 rows.
Connections for id=2:
(2, 1, 1, u'drchuck', 1)
(2, 28, 28, u'cnxorg', 0)
(2, 30, 30, u'kthanos', 0)
(2, 102, 102, u'SomethingGirl', 0)
(2, 103, 103, u'ja_Pac', 0)
20 rows.
```

可以看出，People与Follows表的列和带JOIN子句的SELECT语句执行后得到的结果行。

最后一次选择中我们找到“opencontent”（即People.id=2）的朋友账号。

在最后一次选择的每个“元行”中，前两列来自Follows表，之后是People表5个列中的3个。连接后的每个元行中第2列（Follows.to_id）匹配第3列（People.id）。

14.11 小结

本章全面介绍了Python中数据库的基本使用方法。与Python字典或平面文件相比，编写代码来使用数据库存储数据更加复杂。除非你的应用程序确实需要数据库功能，否则不要轻易使用。数据库的使用优势在于：（1）应用程序需要在大量数据中随机更新一小部分数据；（2）数据量很大，无法用字典来存储，而且需要重复检索信息；（3）在长期运行过程中希望停止或重启，数据可以得以保留，并在下次执行时从中止处继续。

你可以创建单表的简单数据库，以满足多种应用需求。但是，大多数问题都需要多个表与跨表的行之间的链接/关系。为表创建链接时，需要进行周全设计，遵循数据库规范化原则，恰当地运用数据库能力。数据库的使用动机主要需要处理大量数据，有效的数据建模让程序能快速执行，把握住这一点很重要。

14.12 调试

一个常见的Python程序开发模式是与SQLite数据建立联系，执行Python程序，使用SQLite数据库浏览器查看结果。这个浏览器可以快速检查程序是否正常执行。

由于SQLite在同一时刻会防止两个程序对同一数据的修改。例如，如果在浏览器中打开数据库，修改数据库，在尚未按下保存按钮时，浏览器会锁定数据库文件，以防止其他程序访问该文件。具体而言，如果数据库文件被锁定，你的Python程序将不能访问这个文件。

一个解决方法是，在Python尝试访问数据库之前，关闭数据库浏览器或使用浏览器的文件菜单来关闭数据库。这样可以避免数据库锁定导致的Python代码运行失败问题。

14.13 术语

属性：元组的一个值。更常见的提法是列或字段。

约束：告知数据库在表中的字段或行上执行规则。一个常见的约束是保证特定字段上无重复值（即所有值必须唯一）。

游标：执行数据库的SQL命令，并从数据库中获取数据。游标类似于网络连接中的套接字或文件读取的文件句柄。

数据库浏览器：一种无需编写程序，直接与数据库连接并进行操作的软件。

外键：指向另一个表中行的主键的数值键。外键建立了不同表中行之间的关系。

索引：向表中插入行时，数据库软件由于维护需要而产生的额外数据，目的是提高查询速度。

逻辑键：外部世界用来检索特定行的键。例如，在用户信息表中用户的电子邮件地址是不错的用户数据候选逻辑键。

规范化：数据模型设计要保证无重复的数据。每个数据项只存储在数据库的一个位置，其他地方用外键来引用。

主键：每一行指定的数值键，用于当前表中的行与另一个表中的行之间建立引用关系。数据库的默认配置会自动为插入的行赋予主键。

关系：数据库中包含元组与属性的一块区域。更典型的提法是“表”。

元组：数据库中表的一个数据条目，包含一组属性。更典型的提法是“行”。

> 1. 实际上，SQLite在列中存储的数据类型具备一定灵活性，但本章中严格定义数据类型，这样做让这些概念也能适用于其他数据库系统（如MySQL）。↩

> 2. 一般来说，以“如果一切顺利”开头的话，你会发现需要使用try/except。↩

第15章 数据可视化

至此，我们认识了Python语言，学习了Python的基本用法、网络连接，以及使用数据库操作数据等知识。

本章介绍三个完整的应用程序，将之前学到的知识整合起来，对数据进行管理与可视化。你可以使用这些程序代码来解决现实问题。

每个应用程序是一个ZIP文件，可以下载和解压到本地计算机上运行。

15.1 根据地理编码数据创建Google地图应用

这个项目使用Google的地理编码API来清洗用户输入的大学地名，然后将这些数据显示在Google地图上。



从以下地址下载应用程序：

http://www.py4inf.com/code/geodata.zip

首先要解决的问题是，免费的Google地理编码API对每天的请求数量有一定限制。如果数据过多，可能需要在查询过程中多次停止与重启。这里，我们把问题分解成两个阶段。

第一阶段，按行读入where.data文件中的调查数据，通过Google获取地理编码信息，将其存储在geodata.sqlite数据库。在对每个用户输入的地名使用地理编码API之前，我们需要简单检查下输入的行是否有数据存在。数据库具有本地缓存功能，可以对地理编码数据进行缓存，这样就无需向Google重复请求同一数据。

移除geodata.sqlite文件之后，你可以选择任何时候重启这个过程。执行geoload.py程序，依次读入where.data文件的每一行，检查该数据在数据库是否存在。若不存在，调用地理编码API获取该数据库，将其存储在数据库中。

下面是数据库已有一些数据的情况，程序运行结果如下：

```
Found in database  Northeastern University
Found in database  University of Hong Kong, ...
Found in database  Technion
Found in database  Viswakarma Institute, Pune, India
Found in database  UMD
Found in database  Tufts University

Resolving Monash University
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?sensor=false&address=Monash+University
Retrieved 2063 characters {    "results" : [
{u'status': u'OK', u'results': ... }

Resolving Kokshetau Institute of Economics and Management
Retrieving http://maps.googleapis.com/maps/api/
    geocode/json?sensor=false&address=Kokshetau+Inst ...
Retrieved 1749 characters {    "results" : [
{u'status': u'OK', u'results': ... }
...
```

前5个地名已经存在于数据库中，所以它们被跳过。程序会扫描到未检索过的地名，然后开始获取数据。

geoload.py可以随时停止，还有一个计数器用来控制每次运行中地理编码API的调用上限。由于where.data只包括几百个数据，所以无需设置日访问限制。如果数据量很大，需要几天时间的多次运行才能获取所有的地理编码数据，这种情况下就需要设置访问限制了。

载入一些数据到geodata.sqlite之后，你可以使用geodump.py程序对数据进行可视化。此程序会读取数据库，将地名、经度、纬度转换成可执行的JavaScript代码形式，写入where.js文件。

geodump.py运行结果如下：

```
Northeastern University, ... Boston, MA 02115, USA 42.3396998 -71.08975
Bradley University, 1501 ... Peoria, IL 61625, USA 40.6963857 -89.6160811
...
Technion, Viazman 87, Kesalsaba, 32000, Israel 32.7775 35.0216667
Monash University Clayton ... VIC 3800, Australia -37.9152113 145.134682
Kokshetau, Kazakhstan 53.2833333 69.3833333
...
12 records written to where.js
Open where.html to view the data in a browser
```

where.html文件包含了Google地图可视化所需的HTML与JavaScript代码。它读入where.js文件中的最新数据，将其可视化。where.js文件格式如下：

```
myData = [
[42.3396998,-71.08975, 'Northeastern Uni ... Boston, MA 02115'],
[40.6963857,-89.6160811, 'Bradley University, ... Peoria, IL 61625, USA'],
[32.7775,35.0216667, 'Technion, Viazman 87, Kesalsaba, 32000, Israel'],
   ...
];
```

这个JavaScript变量是一个包含列表的列表。JavaScript列表常量的语法与Python非常相似，你应该不会感到陌生。

在浏览器中打开where.html来查看地图。鼠标悬浮在地图标记点上可以看到地理编码API对应的用户输入的地名。如果打开where.html文件看不到数据，你可能需要检查浏览器的JavaScript或在开发者控制台进行排查。

15.2 网络与互联可视化

这个应用程序实现了搜索引擎的一些功能。首先，爬取一部分网页集合，然后实现了一个Google的PageRank算法简化版，确定哪些页面具有高连接度，最后，在这个小网络中对页面等级与连接度进行可视化。

下载和解压这个应用程序：

http://www.py4inf.com/code/pagerank.zip



首先，spider.py程序爬取一个网站，将网站的页面存储到spider.sqlite数据库，记录页面之间的链接。移除spider.sqlite文件之后，你可以随时再次执行spider.py。

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:2
1 http://www.dr-chuck.com/ 12
2 http://www.dr-chuck.com/csev-blog/ 57
How many pages:
```

在程序运行中，我们告知它爬取一个网站，检索两个页面。如果你重启程序，可以让它爬取更多页面，它不会重复爬取数据库已有的页面。程序重启会随机检索未爬取的页面，然后从那里开始。因此，spider.py程序的每一次运行都是累积式的。

```
Enter web url or enter: http://www.dr-chuck.com/
['http://www.dr-chuck.com']
How many pages:3
3 http://www.dr-chuck.com/csev-blog 57
4 http://www.dr-chuck.com/dr-chuck/resume/speaking.htm 1
5 http://www.dr-chuck.com/dr-chuck/resume/index.htm 13
How many pages:
```

同一个数据库中可以有多个起点，在程序中称为”网络“。该爬虫程序随机选择网络中未访问的链接，作为下一个网页进行爬取。

如果要导出spider.sqlite文件内容，spdump.py程序运行结果如下：

```
(5, None, 1.0, 3, u'http://www.dr-chuck.com/csev-blog')
(3, None, 1.0, 4, u'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, None, 1.0, 2, u'http://www.dr-chuck.com/csev-blog/')
(1, None, 1.0, 5, u'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

以上显示了入链的数目、旧的页面排名、新的页面排名、页面id和页面的url。spdumpy程序只显示至少有一个入链的页面。

当数据库里已经保留一些页面数据之后，执行sprank.py程序来实现页面排名。你只需指定页面排名的迭代次数即可。

```
How many iterations:2
1 0.546848992536
2 0.226714939664
[(1, 0.559), (2, 0.659), (3, 0.985), (4, 2.135), (5, 0.659)]
```

再次导出数据库，查看页面排名的更新情况：

```
(5, 1.0, 0.985, 3, u'http://www.dr-chuck.com/csev-blog')
(3, 1.0, 2.135, 4, u'http://www.dr-chuck.com/dr-chuck/resume/speaking.htm')
(1, 1.0, 0.659, 2, u'http://www.dr-chuck.com/csev-blog/')
(1, 1.0, 0.659, 5, u'http://www.dr-chuck.com/dr-chuck/resume/index.htm')
4 rows.
```

sprank.py可以执行多次，它会在每次执行时优化页面排名。你可以执行几次sprank.py，然后用spider.py爬取一些页面，再执行sprank.py来收敛网页排名值。搜索引擎一般会同时运行爬取程序与排名程序。

如果在没有重新爬取网页的情况下再次计算网页排名，你可以用sprest.py程序重置，然后重启sprank.py。

```
How many iterations:50
1 0.546848992536
2 0.226714939664
3 0.0659516187242
4 0.0244199333
5 0.0102096489546
6 0.00610244329379
...
42 0.000109076928206
43 9.91987599002e-05
44 9.02151706798e-05
45 8.20451504471e-05
46 7.46150183837e-05
47 6.7857770908e-05
48 6.17124694224e-05
49 5.61236959327e-05
50 5.10410499467e-05
[(512, 0.0296), (1, 12.79), (2, 28.93), (3, 6.808), (4, 13.46)]
```

对于PageRank算法的每次迭代，它会打印出每个页面排名的平均变化。该网络在初始状态非常不均衡，这是由于单个页面排名值在迭代过程中变化很大。经过一些迭代之后，页面排名开始收敛了。执行prank.py足够长时间之后，网页排名值就会相对稳定。

如果想要对网页排名中当前靠前的页面进行可视化，执行spjson.py程序，读取数据库，将最高连接页面的数据转换为JSON格式，可以在网络浏览器中查看效果。

```
Creating JSON output on spider.json...
How many nodes? 30
Open force.html in a browser to view the visualization
```

在网络浏览器中打开force.html来查看此数据。这是一个自动布局的包含节点与链接的网络。你可以点击和拖拽任一节点，也可以双击节点，查看该节点的URL。

如果重新运行其他工具，重新执行spjson.py，在浏览器中点击刷新，显示spider.json得到的新数据。

15.3 邮件数据可视化

读到这里，你应该还记得mbox-short.txt与mbox.txt这两个数据文件。下面我们将深入分析这些电子邮件数据。

在现实世界中，有时你需要从服务器下载邮件数据，这可能要花费相当长时间，数据可能会存在不一致，充满错误和需要做大量清洗与调整工作。本节介绍的程序是截至目前最复杂的，从服务器下载近1个G大小的数据，然后对其可视化。



下载应用程序代码：

http://www.py4inf.com/code/gmane.zip

这里使用http://www.gmane.org的免费电子邮件列表归档服务。由于该服务提供了电子邮件活动的归档，数据质量高且可搜索，所以在开源项目中非常流行。它的数据API访问政策也比较宽松，没有访问限制，但请不要过度使用，仅获取你需要的数据即可。

你可以在下面的网页中阅读gmane的条款与条件：

http://gmane.org/export.php

使用*gmane.org*数据时需要考虑，在访问服务和长时间运行任务时，有义务添加延时，这一点非常重要。不要滥用这项免费服务，避免累及他人。

使用该软件爬取Sakai电子邮件数据时，可能会产生约1个G的数据，运行需要花费几天时间。下载的压缩包里有一个README.txt文件，提供如何下载一个已爬取的content.sqlite副本的操作指南，无需花费5天时间执行程序来爬取数据。contet.sqlite包含了主要的Sakai电子邮件语料库。如果下载了预先爬行好的内容，你仍然可以执行这个爬取进程，以获取最新的消息。

第一步是爬取gmane归档库。在gmane.py中基础URL是硬编码的，被硬编码为Sakai开发者列表。通过修改基础URL可以爬取其他归档库。如果修改了基础URL，请确保删除content.sqlite文件。

gmane.py文件作为一个“负责任”的缓存型爬虫，有条不紊运行，每秒检索一条邮件信息，这样避免被gmane封掉。它把所有数据存储在数据库，根据需要可以多次中断和重启。数据下载可能需要花费几个小时。因此，程序运行中可能需要重启几次。

gmane.py程序获取到Sakai开发者列表最后5条消息如下所示：

```
How many messages:10
http://download.gmane.org/gmane.comp.cms.sakai.devel/51410/51411 9460
    nealcaidin@sakaifoundation.org 2013-04-05 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51411/51412 3379
    samuelgutierrezjimenez@gmail.com 2013-04-06 re: [building ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51412/51413 9903
    da1@vt.edu 2013-04-05 [building sakai] melete 2.9 oracle ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51413/51414 349265
    m.shedid@elraed-it.com 2013-04-07 [building sakai] ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51414/51415 3481
    samuelgutierrezjimenez@gmail.com 2013-04-07 re: ...
http://download.gmane.org/gmane.comp.cms.sakai.devel/51415/51416 0

Does not start with From
```

该程序扫描content.sqlite，从1开始，直到找到未被爬取的消息的序号，然后开始爬取那条消息。直到爬取到需要的消息序号或者访问到一个不符合消息格式的页面，程序终止。

有时候，gmane.org的消息可能不全，可能是管理员被删除了或消息被弄丢了。如果爬虫停止，可能是它碰到一条丢失的消息。打开SQLite管理器，添加一个丢失的id，其他字段留空，然后重启gmane.py。这样爬取进程就可以继续了。这些空消息在下阶段处理时会被忽略。

一旦爬取了所有消息并将它们存储在content.sqlite，你可以再次执行gmane.py来获取邮件列表上新发布的消息。这听来不错。

content.sqlite的数据缺乏有效的数据模型和未被压缩，显得相当原始。这样做是有意的，它可以让你在SQLite管理器中查看content.sqlite，在爬取过程中调试问题。如果想要对这个数据库进行查询，这可不是个好主意，效率会非常低。

第二阶段是执行gmodel.py程序。该程序从content.sqlite读入原始数据，进行数据清理与建模，生成index.sqlite文件。由于压缩了标题和正文，index.sqlite比content.sqlite文件体积一般要小十倍。

每次执行gmodel.py，它都会删除和重建index.sqlite，允许调整参数和编辑content.sqlite里的映射表，从而控制数据清洗过程。以下是gmodel.py的执行情况示例。每处理250条邮件消息之后打印一行，这样可以观察到一些程序执行情况。该程序会运行一段时间，期间处理近1个G的电子邮件数据。

```
Loaded allsenders 1588 and mapping 28 dns mapping 1
1 2005-12-08T23:34:30-06:00 ggolden22@mac.com
251 2005-12-22T10:03:20-08:00 tpamsler@ucdavis.edu
501 2006-01-12T11:17:34-05:00 lance@indiana.edu
751 2006-01-24T11:13:28-08:00 vrajgopalan@ucmerced.edu
...
```

gmodel.py程序主要是执行一些数据清理任务。

域名.com、.org、.edu和.net被截断成2节，其他域名被分为3节。因此，si.umich.edu处理为umich.edu，caret.cam.ac.uk处理为cam.ac.uk。另外，电子邮件地址全部转为小写。一些@gmane.org地址，如下所示：

```
arwhyte-63aXycvo3TyHXe+LvDLADg@public.gmane.org
```

这样的邮件地址如果与语料库中的真实电子邮件地址匹配，就会被转换为真实地址。

content.sqlite数据库包括两个表，允许域名与个人电子邮件（可能会发生变化）之间进行映射。例如，在Sakai开发者列表中，Steve Githens由于更换了工作，使用以下电子邮件地址：

```
s-githens@northwestern.edu
sgithens@cam.ac.uk
swgithen@mtu.edu
```

我们在contente.sqlite的Mapping表中添加两条数据，这样gmodel.py就可以将3个电子邮件地址映射为一个地址：

```
s-githens@northwestern.edu ->  swgithen@mtu.edu
sgithens@cam.ac.uk -> swgithen@mtu.edu
```

如果多个DNS名需要映射到一个DNS上，你也可以在DNSMapping表中做类似添加。如下映射添加到Sakai数据中：

```
iupui.edu -> indiana.edu
```

这样，所有印第安纳大学的校园账号就可以集中跟踪了。

反复执行gmodel.py来查看数据，通过添加映射让数据更加干净。程序一旦完成，你会得到一个电子邮件的索引版本，即index.sqlite。这个数据库文件用于数据分析非常快。

首先，做两个简单的数据分析："谁发送邮件最多？"和"哪个组织发送邮件最多？"。使用gbasic.py实现：

```
How many to dump? 5
Loaded messages= 51330 subjects= 25033 senders= 1584

Top 5 Email list participants
steve.swinsburg@gmail.com 2657
azeckoski@unicon.net 1742
ieb@tfd.co.uk 1591
csev@umich.edu 1304
david.horwitz@uct.ac.za 1184

Top 5 Email list organizations
gmail.com 7339
umich.edu 6243
uct.ac.za 2451
indiana.edu 2258
unicon.net 2055
```

请注意，与gmane.py和gmodel.py相比，gbasic.py处理数据非常快。虽然它们都在相同的数据上工作，但gbasic.py使用index.sqlite中压缩过和规范化的数据。如果有大量数据需要管理，本节示例应用程序采用的多步处理可能多用了一些开发时间，但在数据探索与可视化时节省了大量时间。

gword.py实现了主题行词频的简单可视化：

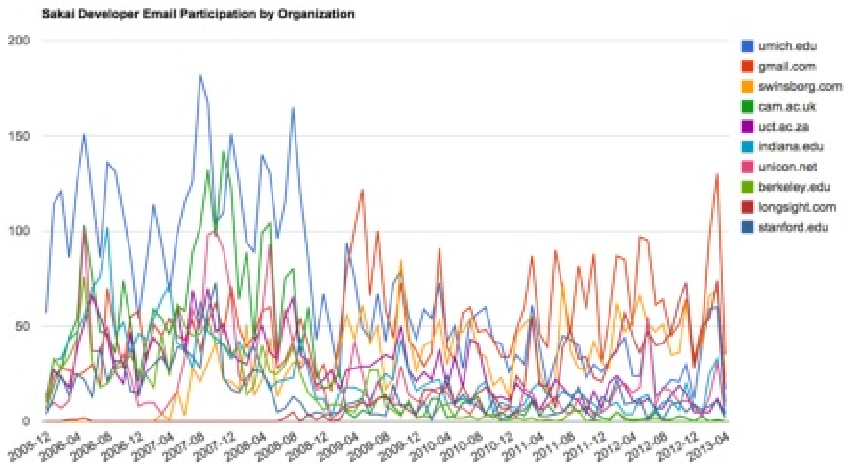
```
Range of counts: 33229 129
Output written to gword.js
```

gword.py执行后生成gword.js文件，通过gword.htm进行可视化，生成一个类似本节开头的词云。

第二个可视化用gline.py生成。它计算了一段时间内某组织的电子邮件参与情况。

```
Loaded messages= 51330 subjects= 25033 senders= 1584
Top 10 Oranizations
['gmail.com', 'umich.edu', 'uct.ac.za', 'indiana.edu',
'unicon.net', 'tfd.co.uk', 'berkeley.edu', 'longsight.com',
'stanford.edu', 'ox.ac.uk']
Output written to gline.js
```

gline.py执行后生成gline.js文件，通过gline.htm进行可视化。



以上是一个相对复杂的高级应用程序，具备一些数据检索、清洗与可视化功能。

第16章 常见任务自动化处理

我们已经学习了从文件、网络、Web Services和数据库中读取数据。Python还可以遍历计算机的所有目录和文件夹，并读取这些文件。

本章中编写的程序会扫描计算机，对每个文件执行某些操作。文件被组织到目录中，也称为“文件夹”。简单的Python脚本既能快速解决简单任务，也能应对目录树或整个计算机上成百上千的文件。

使用os.walk和for循环遍历目录树中的所有目录与文件。这与open方法循环读取文件内容、套接字通过编写循环读取网络连接中的内容以及urllib打开网页读取所有内容等的原理类似。

16.1 文件名与路径

每个运行中的程序有一个“当前目录”，作为大多数操作的默认目录。例如，当打开一个文件进行读取时，Python会在当前目录下寻找这个文件。

os（代表operating system，操作系统）模块提供文件与目录的操作功能。os.getcwd返回当前目录的名称：

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/Users/csev
```

cwd代表当前工作目录。这个示例程序的运行结果是/Users/csev，这是用户csev的当前目录。

类似cwd这样的字符串表示的是路径。相对路径从当前目录开始，绝对路径从文件系统的顶层目录开始。

我们看到的路径都是简单文件名，因此它们是相对于当前目录的。找到一个文件的绝对路径，使用os.path.abspath：

```
>>> os.path.abspath('memo.txt')
'/Users/csev/memo.txt'
```

os.path.exists检查文件或目录是否存在：

```
>>> os.path.exists('memo.txt')
True
```

如果存在，os.path.isdir检查它是否是一个目录：

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

同样地，os.path.isfile检查它是否是一个文件。

os.listdir根据指定目录，返回其下的文件与子目录的列表。

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

16.2 示例：清理照片目录

以前我编写过一个类似Flickr的软件，可以从手机上接收照片，并把它们存储在服务器上。我编写这个软件时Flickr还未出现，当Flickr出现后，我仍然在使用这个程序，用来保留原始照片。

我还会在彩信和电子邮件标题行发送一条简单的文本描述。我将这些消息存在一个文本文件中，放置在照片文件所在的目录下。根据照片拍摄的月、年、日和时间来组织目录结构。以下是照片及其描述的命名示例：

```
./2006/03/24-03-06_2018002.jpg
./2006/03/24-03-06_2018002.txt
```

七年之后，我有了许多照片和标题。这些年我更换过手机，从消息中抽取标题的代码有时会失效，服务器上出现了一些无用的数据。

我想要遍历这些文件，找出哪些文本文件是真正的标题，哪些是垃圾信息，然后删除这些垃圾信息。首先，盘点出子文件夹下有多少文本文件，运行以下程序：

```
import os
count = 0
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           count = count + 1
print 'Files:', count

python txtcount.py
Files: 1917
```

这段代码的关键是Python的os.walk库。当调用os.walk时，指定一个起始目录，它会递归式遍历所有的子目录。“.”表示当前目录，从此处往下查找。当它每遇到一个目录，我们将得到for循环内元组的三个值。第一个值是当前目录名，第二个值是当前目录的子目录列表，第三个值是当前目录的文件列表。

没必要依次查看每个子目录。事实上，通过os.walk来访问每个文件夹。如果想要查看每个文件，编写一个简单的for循环来查看当前目录下的每个文件。如果文件以“.txt”结尾，我们就查看这个文件，并统计整个目录树中以“.txt”为后缀的文件数目。

一旦知道了有多少文件以“.txt”结尾，接下来要自动判断文件的好坏。因此，我们编一个简单的程序，打印出文件及其大小：

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           print os.path.getsize(thefile), thefile
```

现在，不仅仅要统计文件数，我们使用os.path.join创建一个文件名，将目录中的文件名与目录名连接在一起。这里使用os.path.join，而不是用字符串连接。这样做的原因在于：Windows上使用反斜杠（\）来构造文件路径，Linux和Mac上使用正斜杠（/）来构造文件路径。os.path.join知道如何处理这一差异，能够识别当前运行的操作系统，据此选择适合的连接。因此，相同的Python代码在Windows和类Unix系统上都能执行。

一旦得到了带有目录路径的完整文件名，使用os.path.getsize获取文件大小，并打印输出，程序运行结果如下：

```
python txtsize.py
...
18 ./2006/03/24-03-06_2303002.txt
22 ./2006/03/25-03-06_1340001.txt
22 ./2006/03/25-03-06_2034001.txt
...
2565 ./2005/09/28-09-05_1043004.txt
2565 ./2005/09/28-09-05_1141002.txt
...
2578 ./2006/03/27-03-06_1618001.txt
2578 ./2006/03/28-03-06_2109001.txt
2578 ./2006/03/29-03-06_1355001.txt
...
```

扫视一下程序输出，我们注意到有一些文件很短，有些文件非常大，还有一些相同大小的文件（2578和2565）。当打开一些大文件，我们发现它们除了一些通用的HTML标签之外，其他什么都没有。那些HTML从我的T-Moblie手机发送的消息。

```
<html>
        <head>
                <title>T-Mobile</title>
...
```

跳过这个文件，它看起来没有包含有用的信息，随后我们可能做删除处理。

在删除这些文件之前，我们编写一个程序，查找多余一行的文件，并显示文件的内容。不要被2578或2565字符长度的文件所干扰，因为我们已经知道这些文件没有包含有用信息。

程序代码如下：

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           size = os.path.getsize(thefile)
           if size == 2578 or size == 2565:
               continue
           fhand = open(thefile,'r')
           lines = list()
           for line in fhand:
               lines.append(line)
           fhand.close()
           if len(lines) > 1:
                print len(lines), thefile
                print lines[:4]
```

我们使用continue跳过两个“大小不正确”的文件，然后打开其他文件，将读取到文件的内容放到一个Python列表中。如果文件多余一行，打印出文件的行数和前三行内容。

这样一来，程序过滤掉两个大小不正确的文件。假设所有单行的文件是正确的，那么我们得到一些符合要求的数据：

```
python txtcheck.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
3 ./2007/09/15-09-07_074202_03.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/19-09-07_124857_01.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/20-09-07_115617_01.txt
```

但是，文件中还是存在一个或多个令人头疼的模式：有一些三行文件，包含两个空行，之后跟一行文字“发自我的iPhone”，这样的数据仍然存在。因此，针对这个情况修改程序如下：

```
lines = list()
           for line in fhand:
               lines.append(line)
           if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
               continue
           if len(lines) > 1:
                print len(lines), thefile
                print lines[:4]
```

如果是三行文件，程序对其进行检查；如果第三行以特定内容开始，则跳过它。

现在运行这个程序，我们看到还有4个多行文件，但这些文件看起来是合理的：

```
python txtcheck2.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
2 ./2006/03/17-03-06_1806001.txt
['On the road again...\r\n', '\r\n']
2 ./2006/03/24-03-06_1740001.txt
['On the road again...\r\n', '\r\n']
```

纵观程序的整体模式，通过接受或拒绝文件，对结果进行清理。一旦找到“坏”模式，使用continue跳过不符合要求的文件。这样对代码进行修正，找到更多不符合要求的文件模式。

现在，我们准备删除这些文件。这里反转下逻辑，不打印输出剩下的好文件，而是打印出那些不符合要求、准备删除的文件。

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           size = os.path.getsize(thefile)
           if size == 2578 or size == 2565:
               print 'T-Mobile:',thefile
               continue
           fhand = open(thefile,'r')
           lines = list()
           for line in fhand:
               lines.append(line)
           fhand.close()
           if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
               print 'iPhone:', thefile
```

我们得到了一个待删除的候选文件列表，明白了为什么这些文件会被删除。程序运行结果如下：

```
python txtcheck3.py
...
T-Mobile: ./2006/05/31-05-06_1540001.txt
T-Mobile: ./2006/05/31-05-06_1648001.txt
iPhone: ./2007/09/15-09-07_074202_03.txt
iPhone: ./2007/09/15-09-07_144641_01.txt
iPhone: ./2007/09/19-09-07_124857_01.txt
...
```

我们可以检查这些文件，确保没有在不经意间引入错误导致程序结束，或是由于逻辑问题导致一些文件被“错抓”。

当我们对待删除的列表感到满意，对程序做出如下修改：

```
if size == 2578 or size == 2565:
    print 'T-Mobile:',thefile
    os.remove(thefile)
    continue
...
if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
    print 'iPhone:', thefile
    os.remove(thefile)
    continue
```

在这个版本的程序中，我们不仅打印出文件，还使用os.remove移除不符合要求的文件。

```
python txtdelete.py
T-Mobile: ./2005/01/02-01-05_1356001.txt
T-Mobile: ./2005/01/02-01-05_1858001.txt
...
```

出于试验目的，再运行一遍程序，这次不会输出任何结果，这是因为不符合要求的文件已经被移除过了。

如果再次运行txtcount.py，899个不符合要求的文件会被移除。

```
python txtcount.py
Files: 1018
```

在本节中，我们遵循一定处理步骤。首先，使用Python遍历目录和文件来寻找模式；然后，我们在Python的帮助下，确定目录中哪些内容需要进行清理，一旦找到哪些文件符合要求，识别出哪些文件没有用；最后，使用Python进行清理，删除那些没有用的文件。

需要解决的问题可能非常简单，可能仅需要查看文件名，或逐个读入文件，查找文件中存在的模式。有时，你需要读取所有文件，修改其中一些文件。当掌握了os.walk与其他os实用工具之后，这些操作会变得非常简单。

16.3 命令行参数

前面章节中的很多程序都使用raw_input为文件名的输入，从文件读取数据。整个数据处理过程如下：

```
name = raw_input('Enter file:')
handle = open(name, 'r')
text = handle.read()
...
```

对这个程序做一些简化，在Python启动时，通过命令行取得文件名。运行Python程序，提示如下：

```
python words.py
Enter file: mbox-short.txt
...
```

我们可以在Python文件后面附加其他字符串，在Python程序中访问这些命令行参数。下面的程序演示了从命令行读取参数：

```
import sys
print 'Count:', len(sys.argv)
print 'Type:', type(sys.argv)
for arg in sys.argv:
   print 'Argument:', arg
```

sys.argv的内容是一个字符串列表，其中第一个字符串是Python程序的名称，Python文件之后其他字符串是命令行参数。

下面的程序从命令行读取了几个参数：

```
python argtest.py hello there
Count: 3
Type: <type 'list'>
Argument: argtest.py
Argument: hello
Argument: there
```

这三个参数作为三元列表传递到程序中。列表的第一个元素是文件名（argtest.py），文件名之后的其他两个是命令行参数。

我们重写这个程序来读取文件，从命令行参数获得文件名，程序代码如下：

```
import sys

name = sys.argv[1]
handle = open(name, 'r')
text = handle.read()
print name, 'is', len(text), 'bytes'
```

我们把第二个命令行参数作为文件名，在[0]处跳过之前的程序名。打开文件并读取文件内容的代码如下所示：

```
python argfile.py mbox-short.txt
mbox-short.txt is 94626 bytes
```

使用命令行参数作为输入，使得Python程序更易于重用。特别是对仅有一个或两个字符串输入的情况有用。

16.4 管道

大多数操作系统提供命令行界面，也被称为Shell。Shell通常提供文件系统导航与应用启动的命令。例如，在Unix中，cd命令更改目录，ls显示目录的内容，键入诸如Firefox来启动网络浏览器。

从Shell可以启动任何程序，也可以通过Python的管道（pipe）来启动程序。管道是用来表示正在运行的进程的一个对象。

例如，Unix的命令[1]ls –l 通常以长格式显示当前目录的内容。你可以用os.open来启动ls命令：

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

参数是包含Shell命令的字符串。返回值是一个文件指针，这个过程就像是打开一个文件。通过ls进程，readline每次读取一行，或使用read方法一次性得到全部内容：

```
>>> res = fp.read()
```

完成之后，像关闭文件一样关闭管道：

```
>>> stat = fp.close()
>>> print stat
None
```

返回值是ls进程的最终状态。None表示正常结束，没有错误出现。

16.5 术语

绝对路径：从目录树顶层开始，文件或目录所在的位置。无论是否在当前工作目录，都可以访问到文件或目录。

校验：参见哈希算法（hashing）。“校验”这个术语来自于数据的验证需求，当数据在网络上传送或写入到备份介质后再进行读取的过程中，检查数据是否存在被篡改的可能。当数据写入或发送时，发送系统会计算出校验值，一并发送出去。当数据读入和收到时，接收系统会根据接收到的数据，重新计算校验值，与发送来的校验值进行比较。如果校验值不匹配，那么就会认为数据在传输过程中被篡改了。

命令行参数：Python文件名之后命令行中的参数。

当前工作目录：当前你“所在”的目录。在大多数系统的命令行界面，使用cd命令更改工作目录。在Python中，仅使用文件名打开文件，并没有指定路径信息，这时文件必须在当前运行程序的工作目录下。

哈希算法：读取潜在的大量数据，为数据生成一个唯一的校验值。最佳的哈希函数只产生很少的“冲突”。这里的冲突是指哈希函数对两个不同的数据流，产生相同的哈希值。MD5、SHA1和SHA256是常用的哈希算法。

管道：与正在运行的程序进行连接的通道。通过管道，你可以编写程序来发送数据给其他程序，或从其他程序接收数据。管道与套接字类似，但是管道只能用于同一台计算机上程序之间的连接，也就是说不能通过网络进行连接。

相对路径：相对于当前的工作目录，文件或目录所在的位置。

shell： 操作系统的命令行界面。在一些操作系统中称为“终端程序”。在命令行界面中，输入一条命令和参数，然后按下回车键来执行这条命令。

遍历：访问整个目录树、子目录以及子目录的子目录，直到访问到所有的目录。这称之为“遍历目录树”。

16.6 练习

习题 **16.1**：大量MP3文件集合中可能存在相同歌曲的多个副本，存储于不同的目录或者以不同的文件命名。这个练习的目标是找到重复的MP3文件。

1. 编写一个程序，遍历一个文件夹及其子文件夹中的所有以.mp3后缀结尾的文件，并列出相同大小的一对对文件。提示：使用字典，字典的键是从os.path.getsize得到的文件大小，字典的值是文件名与路径名的结合。每遇到一个文件，检查其是否与已知文件的大小相同。如果大小相同，得到一个重复大小的文件，打印该文件大小与两个文件的名称（一个来自哈希，另一个是你正在查看的文件）。
2. 修改之前的程序，用哈希或校验算法查看重复内容的文件。例如，MD5（Message-Digest algorithm 5, 消息摘要算法第五版）接受任意长度的消息，返回一个128位的校验值。不同内容的两个文件返回相同校验值的可能性非常小。MD5的具体内容详见http://wikipedia.org/wiki/Md5。以下代码片段打开一个文件，读入内容，计算校验值。

```
import hashlib
...
            fhand = open(thefile,'r')
            data = fhand.read()
            fhand.close()
            checksum = hashlib.md5(data).hexdigest()
```

新建一个字典，包含校验值作为键，文件名作为值。当计算了校验值，它就作为字典的键存在，有两个文件内容重复，因此打印出字典中的文件和刚才阅读的文件。在图像文件的文件夹下运行程序，结果如下所示：

```
./2004/11/15-11-04_0923001.jpg ./2004/11/15-11-04_1016001.jpg
./2005/06/28-06-05_1500001.jpg ./2005/06/28-06-05_1502001.jpg
./2006/08/11-08-06_205948_01.jpg ./2006/08/12-08-06_155318_02.jpg
```

很显然，我重复提交了相同的照片，没有及时删除之前的拷贝。

> 1. 当使用管道与操作系统命令（如ls）对话时，有一点很重要，了解正在使用的操作系统类型，使用管道打开操作系统支持的命令。↩

附录A Windows平台上的Python编程

本附录介绍在Windows上运行Python的一系列步骤。你可以采用许多种方法做到这一点，但这是一种让事情保持简单的方法。

首先，你需要安装一个程序员专用的编辑器。不要使用记事本或微软的Word字处理软件来编辑Python程序。程序必须是“平面的”文本文件，因此你需要的是一个擅长编辑文本文件的编辑器。

我们在Windows平台上推荐NotePad++，从以下网址下载与安装：
http://sourceforge.net/projects/notepad-plus/files/

从www.python.org网站下载Python 2的最新版本。
http://www.python.org/download/releases/2.7.5/

安装好Python之后，计算机上会出现类似的新文件夹C:\Python27。

新建一个Python程序，从开始菜单运行NotePad++，采用“.py”后缀保存文件。这个练习中，在桌面创建一个py4inf文件夹。文件夹名越短越好，不要在文件夹和文件名中留空格。

编写第一个Python程序如下：

```
print 'Hello Chuck'
```

这里你能做的就是修改成你自己的名字。将程序文件保存在Desktop\py4inf\prog1.py。

在命令行运行程序，Windows不同版本的操作有些许差别。

- Windows Vista与Windows-7: 按下“开始”按钮，在命令搜索窗口输入“command”一词，然后按回车键。
- Windows-XP: 按下“开始”按钮，然后点“运行”，在对话框输入cmd，然后点“OK”。

此时出现一个文本窗口，并提示当前所在的文件夹。

- Windows Vista与Windows-7的文件夹位置是：C:\Users\csev
- Windows XP的文件夹位置是：C:\Documents and Settings\csev

这就是你的“主目录”。现在我们需要进入刚才保存Python程序的文件夹，使用以下命令：

```
C:\Users\csev\> cd Desktop
C:\Users\csev\Desktop> cd py4inf
```

然后输入

```
C:\Users\csev\Desktop\py4inf> dir
```

列出当前文件下的所有文件。当输入dir命令后，你应该看到prog1.py文件了。执行这个程序，只需在命令行输入文件名，然后按回车键。

```
C:\Users\csev\Desktop\py4inf> prog1.py
Hello Chuck
C:\Users\csev\Desktop\py4inf>
```

你可以在NotePad++里编写文件，保存它。然后，切换到命令行窗口，在命令行提示符处，输入文件名再次执行程序。

如果你被命令行窗口搞混了，没关系，关闭它再打开一个即可。

提示：在命令行使用“向上箭头”可以回滚和执行之前输入的命令。

另外，建议在NotePad++的偏好设置中将Tab字符设置为4个空格。这将节省大量由于缩进错误导致的时间花费。

Python程序编辑与执行的更多信息，请参见http://www.py4inf.com。

附录B Mac平台上的Python编程

本附录介绍在Mac上运行Python的系列步骤。由于Mac操作系统已经包含了Python，所以只需学习如何编辑Python文件和在终端窗口执行Python程序。

在Python程序编辑与执行的众多方法中，这仅是我们找到的一种非常简单的方法。

首先，需要安装一个程序员用的文本编辑器。不要使用TextEdit或微软的Word字处理软件来编辑程序。程序必须是“平的”文本文件，因此你需要的是一个善于编辑文本文件的编辑器。

我们在Mac平台上推荐TextWrangler，从以下网址下载与安装：
http://www.barebones.com/products/TextWrangler/

新建一个Python程序，从应用文件夹中运行TextWrangler。

编写第一个Python程序：

```
print 'Hello Chuck'
```

这里你能做的就是修改成你自己的名字。将程序文件保存在桌面，命名为py4inf。文件夹名越短越好，不要在文件夹和文件名中留空格。

在终端窗口执行程序。最简单的方法是点击屏幕后上角的Spotlight图标（放大镜），输入“terminal”（也可输入中文“终端”）启动终端窗口。

这就是你的“主目录”。在终端窗口输入pwd命令查看当前目录。

```
67-194-80-15:~ csev$ pwd
/Users/csev
67-194-80-15:~ csev$
```

必须在包含Python程序的文件夹下执行该程序。我们使用cd命令移动到一个新的文件夹，然后用ls命令列出文件夹下的文件。

```
67-194-80-15:~ csev$ cd Desktop
67-194-80-15:Desktop csev$ cd py4inf
67-194-80-15:py4inf csev$ ls
prog1.py
67-194-80-15:py4inf csev$
```

在命令提示符处，只需输入python命令来执行程序，后面跟程序文件名，然后按回车键。

```
67-194-80-15:py4inf csev$ python prog1.py
Hello Chuck
67-194-80-15:py4inf csev$
```

在TextWrangler编辑Python程序文件，保存它，然后切换到命令行窗口，在命令行提示符处，键入文件名再次执行程序。

提示：在命令行使用“向上箭头”可以回滚和执行之前输入的命令。

另外，建议在TextWrangler的偏好设置中将Tab字符设置为4个空格。这将节省大量由于缩进错误导致的时间花费。

Python程序编辑与执行的更多信息，请参见http://www.py4inf.com。

附录C 贡献

《信息管理专业Python教程》贡献者名单

Bruce Shields审阅了早期书稿，Sarah Hegge, Steven Cherry, Sarah Kathleen Barbarow, Andrea Parker, Radaphat Chongthammakun, Megan Hixon, Kirby Urner, Sarah Kathleen Barbrow, Katie Kujala, Noah Botimer, Emily Alinder, Mark Thompson-Kular, James Perry, Eric Hofer, Eytan Adar, Peter Robinson, Deborah J. Nelson, Jonathan C. Anthony, Eden Rassette, Jeannette Schroeder, Justin Feezell, Chuanqi Li, Gerald Gordinier, Gavin Thomas Strassel, Ryan Clement, Alissa Talley, Caitlin Holman, Yong-Mi Kim, Karen Stover, Cherie Edmonds, Maria Seiferle, Romer Kristi D. Aranas (RK), Grant Boyer, Hedemarrie Dussan,

《思考Python》序

《思考Python》怪史

(Allen B. Downey)

1999年1月我准备讲授一门Java编程入门课程。我已经讲过三遍，但令我很沮丧。课程的失败比例太高，即便是成功的学生也感觉不好，整体的成就感非常低。

我认为问题之一是教材。它们都是大部头，介绍了太多不必要的Java细节，并没有给出足够多的如何编程的高水平的引导。学生都体验到陷阱门效应：他们一开始觉得容易，循序渐进，然后到第五章左右就脱队了。学生得到很多新内容、进度太快了，我得用学期剩下的时间收拾残局。

在第一次课之前的两周时间，我决定写一本我自己的书。我的目标是：

- 保持简短。学生读10页比读50页的效果要好。
- 谨慎使用术语词汇。我尽量少用行话，在第一次使用每个术语进行定义。
- 循序渐进。为避免陷阱门，我将最困难的主题分解成一系列小的步骤。
- 专注于编程，而不是编程语言。我只讲解Java有用的最小子集，其他不涉及。

我需要一个书名，一时兴起，想出了《像计算机科学家一样思考》这么一个名字。第一个版本非常粗糙，但是起作用了。学生真得读进去了，他们能充分理解我在课堂上花时间讲授的复杂主题，有趣的主题和最重要的部分留给他们下去练习。

我使用GNU自由文档许可发布这本书，允许用户复制、修改和分发这本书。

接下来发生的事情是最酷的部分。一位弗吉尼亚州的高中老师Jeff Elkner改编了我的书，把它改编成Python语言。他发给我一份译稿，我通过阅读自己的书来学习Python，这是多么不寻常的体验啊。

Jeff和我修订了这本书，加入了Chris Meyers的一个案例研究。2001年我们还是以GNU自由文档协议发布了《像计算机科学家一样思考：学习Python》。通过绿茶出版社，我出版了这本书，开始在Amazon.com和大学书店出售纸质书。绿茶出版社出版的其他书籍详见 http://greenteapress.com。

2003年我开始在奥林学院教书，第一次讲授Python。与Java的对比是惊人的。学生挣扎更少，学到更多，开发了更多有趣的项目，总体而言乐趣更多。

在过去5年里，我不断修订这本书，纠错和改进一些示例，增加内容，特别是练习。2008年我开始了一个重大版本修订，与此同时剑桥大学出版社一位编辑联系我，对出版此书新版表达出兴趣。多好的时机！

我希望你能喜欢本书，它能帮助你学习编程与思考，至少有那么一点点像计算机科学家。

《思考Python》致谢

(Allen B. Downey)

首先也是最重要的，我要感谢Jeff Elkner，他将我的Java书翻译成Python版本，促使这个项目得以启动，也让我转到了自己最喜欢的语言。

我也要感谢Chris Meyers, 他贡献了《像计算机科学家一样思考》的若干节内容。

我要感谢自由软件基金会提出的GNU自由文档许可协议，这让我与Jeff和Chris共事成为可能。

我还要感谢《像计算机科学家一样思考》的Lulu网责任编辑。

我要感谢所有参与到本书早期版本编写的学生，所有提交勘误与建议的贡献者名字显示在附录中。

我要感谢我的妻子Lisa为本书的付出，还有绿茶出版社和其他所有提供帮助的人。

Allen B. Downey Needham MA

Allen Downey是富兰克林.奥林工程学院计算机科学副教授。

《思考Python》贡献者名单

在过去几年间，有100多位眼光敏锐、有想法的读者给出建议和提交勘误。他们对这个项目的贡献与热情，成为最大的帮助。

每个人的贡献细节参见《思考Python》正文。

Lloyd Hugh Allen, Yvon Boulianne, Fred Bremmer, Jonah Cohen, Michael Conlon, Benoit Girard, Courtney Gleason and Katherine Smith, Lee Harr, James Kaylin, David Kershaw, Eddie Lam, Man-Yong Lee, David Mayo, Chris McAloon, Matthew J. Moelter, Simon Dicon Montford, John Ouzts, Kevin Parks, David Pool, Michael Schmitt, Robin Shaw, Paul Sleigh, Craig T. Snydal, Ian Thomas, Keith Verheyden, Peter Winstanley, Chris Wrobel, Moshe Zadka, Christoph Zwerschke, James Mayer, Hayden McAfee, Angel Arnal, Tauhidul Hoque and Lex Berezhny, Dr. Michele Alzetta, Andy Mitchell, Kalin Harvey, Christopher P. Smith, David Hutchins, Gregor Lingl, Julie Peters, Florin Oprina, D. J. Webre, Ken, Ivo Wever, Curtis Yanko, Ben Logan, Jason Armstrong, Louis Cordier, Brian Cain, Rob Black, Jean-Philippe Rey at Ecole Centrale Paris, Jason Mader at George Washington University made a number Jan Gundtofte-Bruun, Abel David and Alexis Dinno, Charles Thayer, Roger Sperberg, Sam Bull, Andrew Cheung, C. Corey Capel, Alessandra, Wim Champagne, Douglas Wright, Jared Spindor, Lin Peiheng, Ray Hagtvedt, Torsten Hübsch, Inga Petuhhov, Arne Babenhauserheide, Mark E. Casida, Scott Tyler, Gordon Shephard, Andrew Turner, Adam Hobart, Daryl Hammond and Sarah Zimmerman, George Sass, Brian Bingham, Leah Engelbert-Fenton, Joe Funke, Chao-chao Chen, Jeff Paine, Lubos Pintes, Gregg Lind and Abigail Heithoff, Max Hailperin, Chotipat Pornavalai, Stanislaw Antol, Eric Pashman, Miguel Azevedo, Jianhua Liu, Nick King, Martin Zuther, Adam Zimmerman, Ratnakar Tiwari, Anurag Goel, Kelli Kratzer, Mark Griffiths, Roydan Ongie, Patryk Wolowiec, Mark Chonofsky, Russell Coleman, Wei Huang, Karen Barber, Nam Nguyen, Stéphane Morin, and Paul Stoop.